THE UNIVERSITY OF ALBERTA

Microcomputer Based Computer-Assisted Learning System: CASTLE

by

© Robert William Thomas Garraway

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF Master of Education

Department of Educational Psychology

EDMONTON, ALBERTA

SPRING 1983

Dedication

To my wife, Bahiya, and my children, Naisan and Yasmin.

Abstract

The purpose of this study was to investigate the extent to which a sophisticated CAI/CAL system could be implemented on a typical microcomputer system currently found in the schools. A method for comparing CAL languages was devised and used to rank five common CAL languages. NATAL, Canada's national authoring language, was found to have the highest rank and so was used as the basis for the design of a microcomputer based CAL system.

The new system, named CASTLE (*C*omputer-*A*ssisted *S*tudent *T*utorial *L*earning *E*nvironment), was primarily designed for trained CAL authors and researchers, but has features that could assist the beginning author in creating CAL lessons and courses. The design specifications for the CASTLE Language and the CASTLE Support System were defined.

A subset of the CASTLE Language and Support System was developed on a Commodore CBM 8096 microcomputer system. The CASTLE system software was written in COMAL-80. Techniques of incremental compilation in an interactive and interrogative environment were used. The completed system was evaluated and recommendations made for further research and development.

Acknowledgements

I would particularly like to thank my supervisor, Dr. Steve Hunka, for his guidance, patience, and encouragement throughout the duration of the research. I also greatly appreciate the comments and assistance given by my committee members. Alan Davis, of the Division of Educational Research Services, was especially helpful in increasing my understanding of computer software systems design.

The financial assistance and facilities of the Division of Educational Research Services made this project possible.

Finally, I wish to thank my children for their patience and my wife for the long hours of entering this thesis text into the computer.

Table of Contents

List of Tables

List of Figures

I. Introduction

> In 1981, the debate over whether or not microcomputers will substantially impact education has all but vanished. Tens of thousands of micros have made their way into the schools, and discussions have at last turned to the planning needs ... the preparation of software and courseware (software which teaches) for use with microcomputers. (Roblyer, 1981, p. 47)

With this influx of microcomputers into the schools there has been a resurgence of interest among teachers and school administrators in computer-assisted instruction (CAI). Most have been scanning the market place for good courseware for use with their students. Many have expressed an interest in writing their own CAI programs. Unfortunately good courseware for microcomputers is rare and specialized CAI software systems almost non-existent or of limited sophistication.

A. The Development of a Microcomputer Based CAI System

The only CAI language widely supported on todays microcomputers is PILOT. It is relatively easy to learn and use, but does not support modern structured programming techniques, built-in performance recording or student restart points, and has only one type of answer analysis. Various courseware development systems designed for use by inexperienced authors have appeared on the market. Many of these are useful for implementing simplistic CAI but they lack many of the features considered 'standard' on main-frame CAI systems. Even the expensive PASS system developed by Bell & Howell, though having many fine

features, lacks expansibility, locking the experienced author into set instructional strategies.

The purpose of this thesis is to investigate the extent to which a sophisticated CAI system can be implemented on a typical microcomputer system currently found in the schools. This system should have features useful to researchers in CAI. These might include flexible specification by the author of the information to be included in performance records; and author developed response comparison functions, input edit functions, and general functions.

CAI has been around for more than twenty years. This experience provides a large body of knowledge and research upon which to build CAI systems and courseware not only for the microcomputers presently in the field, but for the more powerful ones now being developed.

B. The Evolution of CAI

Two Courseware Philosophies

Roblyer (1981) suggests that two courseware philosophies have evolved during the last two decades of CAI development. One is called the "PLATO model" after the PLATO CAI project at the University of Illinois, and the other, the "Stanford/CCC model" after the work of Suppes which began at Stanford University and was later marketed by the Computer Curriculum Corporation (CCC). The "PLATO model" is

primarily used for mainline instruction where the computer supplies "highly-interactive 'conversational' tutorials which simulate the presentation of an excellent classroom teacher." (Roblyer, 1981, p. 48) On the other hand, the "Stanford/CCC model" presents drills in basic skills in brief daily sequences "intended to supplement and reinforce what has been previously taught in the classroom." (p. 48)

Roblyer outlines three areas where the two models differ:

> 1. *Learner control.* Instructional activities in most PLATO materials usually emphasize student ability to structure his or her own lesson path by using a menu (index), on-line glossary or other "Help" resource, remedial feedback and loops, and control of frame movement, both forward and back. Many materials based on the Stanford/CCC model, on the other hand, consist of highly-machine-controlled drill exercises, with most other learning activities off-line.
>
> 2. *Feedback.* Many intricate answer-judging features and capabilities were built into the PLATO author-language, to allow feedback tailored to several different kinds of student answers. Positive reinforcement for correct answers is usually given, either in the form of verbal feedback ("Good work, Morley"), or an animated creature of some kind. If the answer is wrong, explanation is supplied matched to the type of response (e.g., misspelling, concept error, or unexpected response). Many Stanford/CCC materials have no response at all to the student if the answer is correct, and often only a "No, try again," if the answer is wrong the first time.
>
> 3. *Graphics and animation.* Again, the emphasis placed on these features may be seen in the number of tools and commands in the PLATO author language for developing graphic displays. Most PLATO lessons contain animated creatures or diagrams of some kind. However, few graphics appear in the CCC materials. (p. 48)

PLATO authors could write courseware that conformed to the

Stanford/CCC model, but they rarely do, tending instead to utilize the richness of the facilities offered by the PLATO environment. Roblyer concludes from examination of research with these two models that **both** are very effective. The genesis of these models goes back more than two decades.

The Early Years

Burns and Bozeman (1981) have outlined some of the events of the early years of CAI development.

The dawn of CAI was in the late fifties when the computer industry began exploring its use for training their own personnel. IBM took the lead, with DEC, CDC, and Hewlett-Packard soon following. The early sixties saw many universities begining research and development projects in CAI. Educational CAI, stimulated by U.S. federal funds, was a natural combination of the emerging computer technology and the programmed instruction movement.

One of the earliest university projects was the Stanford project at the Institute for Mathematical Studies in the Social Sciences. It was led by Patrick Suppes. Among the first to venture into the puplic schools with CAI, by 1963 they had developed a small tutorial system in elementary mathematics and language arts. Their second system concentrated on the drill and practice mode of CAI. During 1967/68 they were providing 3000 students with daily lessons in initial reading, mathematics, and spelling.

In 1967, the Stanford group founded the Computer Curriculum Corporation (CCC) as a commercial outlet for their research. Their courseware was implemented on a Data General mini-computer in machine language, and presented on up to 96 simple CRT terminals. (Hallworth and Brebner, 1980, p. 45) Drills in mathematics, reading, and language arts intended for basic skill development and maintenance were provided for levels K to adult. They were particularly successful with the culturally and academically disadvantaged.

The PLATO (Programmed Logic for Automatic Teaching Operations) project began in 1960 in the Coordinated Science Laboratory at the University of Illinois. After a seven year developmental phase, during which over 300 programs were written, the University founded the Computer-Based Educational Research Laboratory. Here the PLATO III and, in the seventies, the PLATO IV systems were developed.

In 1967, the Waterford, Michigan School District commenced their INDICOM project that saw the development of teacher-authored CAI courseware in eleven content areas for grades K to 12. "A system approach to curriculum creation accommodated behavioral objectives specifications, instructional sequencing, and procedures for evaluating model effectiveness." (Burns & Bozeman, 1981, p. 33)

The National Science Foundation funded research projects in the late sixties to investigate what, besides drill and practice, could best take advantage of the

computer in the educational environment. Two of these projects, the Huntington Project at the State University of New York and Project SOLO at the University of Pittsburgh, examined the simulation, games, and problem solving modes of CAI. (Freiberger, 1981)

The Active Seventies

The history of CAI in the seventies has been discussed in some detail by Hallworth and Brebner (1980).

One of the most active CAI systems during the seventies was the IBM 1500 system. At one time it was in use by twenty-five centers. The system, along with the Coursewriter II CAI language, was developed by IBM in the mid-sixties with the assistance of Suppes' group at Stanford University. It supported up to thirty-two multi-media terminals. The 1500 System Users Group shared courseware and user developed system enhancements such as powerful answer analysis functions and graphics construction sub-systems.

Typical of the 1500 users of that period was the CAI group of the Division of Educational Research Services at the University of Alberta. They operated a 1500 system from 1968 until its withdrawal from service by IBM in April 1980. Full tutorial courses in such areas as medicine, statistics, and CAI authoring were developed during this period as well as support programs for course documentation and student performance analysis.

> During the last five years of operation of this computer system, approximately 26,000 student-hours

> of instruction per year were offered, mainly through courses which formed the primary source of instruction for the student. Some courses averaged 70-80 hours to complete by the average student.
>
> Of course, the IBM 1500 system had a number of hardware disadvantages as well as cost disadvantages .... [It] had no adequate remote capability and all terminals had to be within about 1000 feet of the central facility. The response time to 22 terminals could be degraded by poor programming practices, and mainframe and disk memory was limited. Nevertheless, the software system as it was designed and enhanced by the Division for the support of instruction, for its time was superb, and can only be currently matched by the far more expensive PLATO system. (Hunka, 1981)

Also during the seventies, the PLATO IV system continued to evolve and expand. It was commercially marketed by Control Data Corporation (CDC). PLATO IV can support several hundred terminals at remote locations. Some may be multi-media terminals. Its relatively facile authoring language, TUTOR, has been used to produce a very large base of CAI lessons. The University of Illinois system now services over 4000 students per semester. (Burns & Bozeman, 1981)

> Among the general conclusions which can be drawn from observing the PLATO project in operation, both in schools and at the central site, the one which stands out is that this is a very active project, strong in research and with many creative CAI authors who are producing excellent courseware in many subject areas at all levels. (Hallworth & Brebner, 1980, p. 28)

Other systems of note during the seventies was that produced at the Ontario Institute for Studies in Education (OISE) and the TICCIT (Time-shared, Interactive, Computer Controlled Information Television) system created by the MITRE Corporation. OISE developed their own authoring

language, CAN, and produced courseware in remedial mathemaics that is used by many community colleges. TICCIT was one of the first systems to explore the use of colour graphics in CAI.

The Effectiveness of CAI

Forman (1981), after doing an extensive survey of the literature, draws the following generalized conclusions regarding the effectiveness of CAI in the learning process:

> 1. The use of CAI either improved learning or showed no difference when compared to traditional classroom approaches.
>
> 2. The effect on achievement occurred regardless of the type of CAI used, the type of computer system, the age range of the students, or the type of instrument used to make the measurements.
>
> 3. When CAI and traditional instruction are compared, equal or better achievement using CAI is obtained in less time.
>
> 4. Students have a positive attitude towards CAI, frequently accompanied by increased motivation, attention span, and attendance in courses. (p. 46)

As Hallworth and Brebner (1980, p. 102) point out, these established CAI projects made use of a technology that was often more than ten years old. Schools, for the most part, have been reluctant to establish CAI projects because of the high initial cost of the hardware required. However, the new micro technology has reduced costs tremendously.

C. The Microcomputer Revolution

Mankind has witnessed the extrordinary growth and development of the computer industry since the Second World War. The ENIAC computer of 1946 took up 15000[1] sq. ft. of space, contained 18,000 vacuum tubes, cost several million dollars, and kept a horde of technicians busy just to maintain its use for ten minutes a day. Todays microcomputer takes about two square feet of space, costs about a thousand dollars, requires little maintenance, and has far more computational power. (Johnson, 1981)

These changes were the result of various stages of development within the electronic industry. The first generation of computers was based on the technology of the vacuum tube. Succeeding generations were based on the transistor, the integrated circuit, and now the large scale integrated (LSI) circuit or silicon chip. Each generation brought computers that were faster, had greater capacity and power, were more reliable, and were less expensive. Johnson (1981) cites a recent computer company's advertisement which states that if the auto industry had progressed at the same rate as the computer industry over the last thirty years, a Rolls Royce would cost $2.50 and would get 2,000,000 miles per gallon.

A computer has, of course, many components: fast access memory, slower access large scale memory, input/output

----

[1] Johnson actually states 1500 sq. ft., but this is obviously too small. Rice (1976) gives 15000 sq. ft., which is more likely.

controls, input/output devices, and a central processing unit (CPU). The CPU itself has many complex components. INTEL in 1969 was the first to put a CPU on a single silicon chip. This was the birth of the microprocessor. This first microprocessor was slow and had only a four bit word length. It was soon succeeded by the INTEL 8080 eight-bit microprocessor which was much faster and more reliable. Other chip manufacturers quickly had their own microprocessors on the market, supported by new memory and input/output control chips.

Hallworth and Brebner (1980) described the next stages of the revolution:

> The availability of silicon chips is a necessary but not sufficient condition for the development of a microcomputer. ... The appropriate chips must be assembled, generally on boards, and interfaced to supply all the necessary hardware functions of the microcomputer; and a power supply must be added. The whole must then be provided with a software system, for which it is also necessary to provide terminal handlers, and preferably at least one high level language.
>
> The development time required to produce such a microcomputer meant that the first did not appear until January 1975. This was the Altair 8800, which was produced in kit form. During 1975 and 1976 most microcomputers were sold in this form, as hobby computers, intended for people having some prior knowledge of engineering and computing. Such microcomputers were obviously not appropriate for most educators.
>
> However, in April 1977 Commodore announced the PET 2001 microcomputer, a fully operational turnkey package, designed not for the hobbyist but for the general consumer. This represented a dramatic change: it was the first microcomputer that had immediate appeal to the educator. Within a year, PET's had been purchased by a large number of schools in the U.S. (pp. 111-112)

Each year more models have appeared on the market with an ever expanding list of support peripheral devices: such as floppy disk units, printers, light pens, graphics tablets, and speech synthesizers. And each year thousands of more microcomputers have found their way into the schools of the industrialized world.

D. The Wide Spread Use of Computers in Education

Forman (1981), citing the 1980 survey by Chambers and Bork of a selected sample of 974 U.S. school districs, found:

> Approximately 90% of all school districts responding are now using the computer in support of the instructional process. Most computers are leased or owned by districts and large computers are more in evidence than are micros and minis which the study found to be equal in popularity. It was also found that the most popular applications in order of usage are the teaching of computer languages, computer assisted learning, data processing applications, using the computer as an instructional aid, and using it for guidance and counselling applications. (p. 64)

From this same report there was noticed a dramatic increase in the percentage of districts making use of computers directly in instruction: 13% in 1970 to 74% in 1980 with 54% as CAI/CAL. 94% of the reporting districts expect to be using computers by 1985 with 87% in direct instruction and 74% as CAI/CAL.

Klassen and Solid (1981) reported on a March 1981 survey by the U.S. Department of Education, National Center for Educational Satistics entitled *Student Use of Computers in Schools*. It found that in about half of the school

districts of the U.S. students had access to microcomputers or computer terminals. There were over 31,000 microcomputers in the schools and almost 21,000 terminals.

The most widely used CAI systems have been those based on PLATO and those of the Computer Curriculum Corporation (CCC). Hallworth and Brebner (1980) report that in 1980 there were nine PLATO systems in existence, all but the one at the University of Illinois built by Control Data Corporation (CDC). There are systems at the Universities of Florida, Deleware, Alberta, and Quebec; as well as two in CDC's main offices in Minneapolis and one each in Europe and South Africa. Over 100 colleges, universities, medical facilities and schools, public schools, government agencies, and businesses in both Canada and the U.S. access the more than 6000 hours of student-tested instruction in over 100 subjects on the PLATO IV systems. (Menashian, 1981)

"CCC has provided the major portion of all CAI which has been used in schools on a regular basis." (Hallworth & Brebner, 1980, p. 46) One-third of the installations are in Texas with other systems in California, Seattle, Mississippi, New York, Illinois, New Mexico, Pennsylvania, and Arizona.

In 1973 the Minnesota Educational Computer Consortium (MECC) was formed by the University of Minnesota, the States University System, the Community College System and the State Departments of Education and Administration. Hallworth and Brebner (1980, p. 76) report that MECC's division of

Instructional Services manages and operates a state-wide time-sharing computer network that supports approximately 2000 mostly simple Teletype terminals.

> Applications on the network range from simple drill programs for skill improvement to complex simulations of historical events and guidance systems. In all, over 950 instructional programs have been implemented or developed and are available to all users of the network to supplement curricula at elementary, secondary and college levels. (p. 77)

As more and more microcomputers came on the market, MECC sponsored a project which evaluated fifteen of the then current models. The result was a volume purchase agreement with Apple Computers that has seen more than 1000 Apple II microcomputers enter the schools there. The educational programs developed for the Apple II by MECC have been purchased and used by schools across North America.

The Ministry of Education of the Province of British Columbia has been operating a pilot project on the use of microcomputers in the schools. It is based on the MECC model. Initially fifty of the provinces seventy-five school districts submitted proposals to the project. Of these, twelve were selected and 100 Apple II microcomputers were distributed to the schools during the summer of 1980. Teachers on the project were given their initial training on these computers at special summer courses at the Universities of Victoria and British Columbia.

The mid-project formative evaluation indicated that "the single most critical issue in the use of microcomputers in the schools of B.C. was the acquisition, development, and

sharing of quality CAI materials relevant to the B.C. curriculum." (Forman, 1981, p. 16) The subjective findings of the final report showed that "all of the [12] coordinators and the majority of the teachers ranked CAI as the most important use of the microcomputer, with courseware development ranking second." (p. 29)

Lindsay, Marini, and Lancaster (1980) reported that The Department of Special Education of The Ontario Institute for Studies in Education carried out a survey of the School Boards of that province in May of 1980 to determine the frequency and the nature of their microcomputer applications. Ninety-five of the 182 boards surveyed responded to the questionnaire sent. They represented 84.5% of the total school population.

> Over 50% of the responding boards, indicated that they currently had at least one microcomputer in use. For these boards, the average number of microcomputers was 13.6, the median 7, the range 1 to 79. In total, 652 micros were reported to be in use in Ontario schools, 624 of these were designated exclusively for instruction. These 624 computers were distributed across a total of 157 different applications. (p. 27)

65.8% of these micros were Commodore PET's, 16.8% were TRS-80's, and 5.8% were Apple II's. 91% of the applications had been in use for only one year or less. Almost 61% of all micros were used in grades 10 to 13, and 51% were being used for teaching computer programming. The following types of programs were reported in use:

Introduction of New Material ... 38.3%

Drill and Practice 31.8%

Simulation 13.6%

Games 5.9%

Other 10.4%

(p. 29)

Petruk (1981) reported the findings of a survey conducted for Alberta Education in the fall of 1980:

> The results showed that nearly 12% of Alberta schools now have one or more microcomputer. The majority of the units are Commodore PET (45%), Apple II (31%) and Radio Shack TRS 80 (19%). They appear to be uniformly distributed across all grade levels. The most frequently reported uses of the microcomputer involved the teaching of computer literacy and computer assisted instruction. ...
>
> While a relatively small number of schools reported that they had no interest in introducing microcomputers into their school, the majority of schools that do not now have a microcomputer are anticipating getting one or more in the future. (p. 18)

The use of microcomputers for computer assisted instruction, as mentioned in the above quotation, may need clarification. It usually implies the use of courseware written in BASIC, such as drills, games, or simulations of an educational nature; but does not imply a supporting CAI system of student records or the recording and analysis of student performance. Many writers are now referring to this form of CAI as "simplistic CAI" and contrast it to advanced CAI systems such as PLATO.

In October 1981 the Minister of Education for the Province of Alberta announced the bulk purchase by his department for resale to the schools of 1000 of the Bell & Howell version of the Apple II with disk drives, printer,

and colour monitor. He understood that there were about 1000 microcomputers in the schools of Alberta and hoped that this number would triple over the following eighteen months. (King, 1981)

Wise (1981) reported that market research firms project that the annual sales of microcomputers to educational institutions in North America will be greater than 250,000 by 1985.

E. Issues Blocking CAI in Schools

Forman (1981), after reviewing the literature, lists the following as "factors which researchers have identified as being impediments to the exploration of the full potential of the computer in education:"

1. Insufficient funding from the appropriate sources to support the original purchase of hardware, software, courseware, and to establish the necessary support services for the successful integration of the technology into the education system. (Chambers & Bork, 1980; Kearsley, 1976; Luehrmann, 1980; Moursund, 1979; Splittgerber, 1979)

2. The primitive state of the art in which there is a confusing diversity of languages and hardware systems. (Chambers & Sprecher, 1980; Kearsley, 1976)

3. CAI materials that are poorly constructed, largely undocumented, and able to run on only the equipment for which they were written. (Chambers & Sprecher, 1980; Kearsley, 1976)

4. Lack of knowledge among educators as to how to effectively use CAI materials and the computer in the learning situation, particularly at the moment when limited financial resources restrict the number of systems available per classroom. (Chambers & Sprecher, 1980; Kearsley, 1976; Moursund, 1979)

5. The attitude among teachers, familiar with and comfortable using tried and tested methods, that the computer is not a tool but an intelligent machine destined to replace them as teachers. (Chambers & Sprecher, 1980; Clement, 1981; Kearsley, 1976; Splittgerber, 1979; Travers, 1981)
(p. 60)

Also, producing good courseware is a difficult and lengthy task. Gleason (1981) cited by Forman (1981) states:

> It involves careful specification of objectives, selection of programming strategies, detailed analysis of content structure and sequence, development of pretests and posttests, preliminary drafts, revisions, trials, validation, and documentation. This is a very time-consuming and expensive process, well beyond the capacity and resources of individuals and even small groups of teachers. (p. 69)

Even with the extensive author tools provided by the PLATO IV system, "inexperienced authors require an average of 237 hours to produce each hour of student material while the rate falls to an average of 26.4 hours for those with experience." (Hallworth & Brebner, 1981, p. 18)

Hardware costs of main-frame CAI computer systems have often been cited as a factor retarding the introduction of CAI into the schools. For example the net monthly fee charged by IBM to the Division of Educational Research Services at the University of Alberta in 1979/80 to rent and maintain an IBM 1500 CAI system with seventeen multi-media terminals was $14,176.23. That would be $833.90 per station per month.

During the period 1980 to 1982 the University of Alberta installed a CDC PLATO IV system with eighty terminals. The total hardware cost for this system was

$2,420,720. If amortized over a five year term the per terminal cost would be $504.32 per month. This does not include maintenance costs or the salaries of support personel. Multi-media terminals would cost an additional $5605 per terminal. Again, when amortized this would add another $93 per terminal per month.

The unit cost of microcomputers is considerably lower. As an example, the Alberta Department of Education is selling to the schools the Bell & Howell version of the Apple II with 48K of memory, dual disk drives, a Panasonic colour monitor, an integer card, and a clock for $4255.28. Amortized over five years, the monthly rate would be $70.92.

As a further example of reduced unit cost, the Commodore microcomputers can be linked together to form a small system that shares input/output devices. Commodore also offers a special purchase deal to educational institutions; purchase two microcomputers and get the third one free. An example nine station system would be priced as follows:

9 - CBM 8032 micros (3 for 2)	$13,948.20
1 - 8050 1Mbyte duel disk unit	2,464.65
1 - 4022 tractor feed printer	1,290.21
1 - 8 channel MUPET system	2,245.00
1 - MUPET spooled printer channel	700.00

The total cost would be $20,648.06 or $2294.23 per station. Again, if amortized over five years the monthly cost per station would be $38.24. It should be noted, however, that

these microcomputers do not have the sophisticated CAI systems software or courseware that comes with the main-frame computers. But with such low per unit costs, it is understandable why so many microcomputers are now entering schools.

F. The Development of Computer Languages

The earliest computers were very difficult to program. Instructions were entered in the binary code of the machine by means of panels of switches and the use of "patch cords." Later, each of the computer's instructions were given mnemonic three character names and a computer program was developed for each machine that could assemble these instructions into the binary machine code. This was assembler language programming. Each computer had a different assembly language. However, in 1956 a new breed of programming languages appeared. These were called high level languages because they were not usually limited to use on one type of machine and they used natural language key words. Some of these early high level languages and their year of introduction are: FORTRAN (1956), COBOL (1960), ALGOL60 (1960), and LISP (1960). Since then many other general purpose and specialized high level languages have been developed. Among the specialized languages have been those dedicated to computer-assisted instruction, such as Coursewriter, TUTOR, CAN, and NATAL.

As computers and their languages progressed, a class of computer software known collectively as operating systems was also evolving. The operating system software was designed to take care of the general housekeeping chores of the computer system. It helped the programmer to use all of the resources supported at a particular installation.

This same evolution of computer software and languages has been repeated for microcomputers over their brief history. The first micros were programmed in machine language using panel switches, then came assemblers for the various microprocessors, followed by the high level language BASIC. FORTRAN, COBOL, Pascal, FORTH, APL, and other languages have now been implemented on many of the micros, but not, as yet, any of the more sophisticated CAI languages.

Commenting on the use of microcomputers by teachers for the development of courseware, Hunka (1981) states:

> For the most part, the software available on microcomputers requires far greater understanding of the structure of computers than that which is required for effective use of a large-scale computer. The user of the micro must be able to take care of a far greater number of housekeeping tasks during the development and execution of program code. ...
>
> The implementation of instruction based upon those factors which we already know enhance classroom instruction, cannot always be easily done in BASIC. BASIC was never designed to provide the vehicle for the development of instructional courseware. Many researchers in education and computing science are striving to develop the kind of programming languages which are required, and at least a dozen can be easily identified, including the NATAL and CAN languages developed in Canada. ... There are other programming languages which are

available on some microcomputers, these include FORTRAN, Pascal, FORTH, and APL. But again, although these languages have certain marked advantages over BASIC, they were not designed to be used for the wide range of procedures required in executing an instructional program. (p. 9)

G. CAI as a Solution to Educational Problems

In an examination of the present state of the quality of education, Hallworth and Brebner (1980) made the following assessment:

> The industrialized societies of Western Europe and North America have organized their educational systems upon allegedly homogeneous classes of students under the charge of one teacher. The system ... depends on the use of printed texts. Moreover, it has fulfilled its purpose, in ensuring universal education to a level needed by industrialized societies.
>
> However, these very societies are now in the process of entering a technological era and are discovering the need for higher levels of education in their citizens. One consequence is pressure upon schools, and upon teachers, to insure higher levels of achievement in their students. A solution is to provide opportunities for individualized learning. Most professionals in the field of education would now subscribe to the need for individualization, for open classrooms and a well structured curriculum.
>
> It is questionable, however, whether such objectives can be adequately achieved without the introduction of a new dimension into the organization of learning and instruction. Studies ... have suggested that, with the present arrangements, a teacher is unable to devote more than one or two minutes per day of individual attention to each student. ... The problem with the present system is not that the teacher is not teaching; it is that, for lack of individual attention, students are not learning. ...
>
> It is suggested that a new dimension can now be introduced by making use of the information storage, processing, and distributing capabilities of computers. ... Appropriately used, CAI can help ensure that students will receive a greater degree

> of individual attention than they receive at present, that they will learn more quickly and effectively, and that they will have a positive attitude towards future learning. (pp. 183-185)

Hunka and Romaniuk (1974) and Forman (1981) have suggested a number of ways in which CAI can assist in solving some of the problems found in todays classrooms. By allowing students to progress at different rates and use different methods, by providing students with remediation or enrichment, and by giving students immediate feedback and systematic reinforcement, CAI is able to individualize instruction. The use of CAI facilitates flexible scheduling by allowing many students to take different courses at the same time, or to take courses outside regular hours. Courses are sometimes not offered for lack of qualified instructors. CAI could help overcome this by offering courses developed by qualified outside agencies. It can free teachers from many mundane and time-consuming tasks, and allow them to devote more time to the personal, human considerations of their students. Good CAI can provide systematically sequenced and carefully prepared, tested, and revised instruction. Finally, as has been mentioned on page 7, CAI has been shown to be able to motivate students of all levels of interest and ability, and to actively involve them in their learning.

H. The Characteristics of CAI

In the traditional classroom a teacher presenting a course must consider what content to present in each lesson, what teaching strategies to use, and how to monitor each student's progress in achieving the course objectives. These characteristics must also be present in a good CAI system.

Hunka and Romaniuk (1974) have outline some of the characteristics of the better main-frame CAI systems:

1. Curriculum material is stored in one of the memory systems of the computer. Curriculum material may include textual material (with differing character sets ...), graphic material ..., visual material such as static pictures ..., and audio messages....

2. The lesson material is presented to a student via a computer terminal [usually on a] television screen ... with an associated keyboard, light pen [or a touch panel], a photographic projection device, and an audio play/record unit. ...

3. The curriculum material is presented to the student following a precise instructional logic defined by the author of the course .... The instructional logic is also stored in one of the computer's memory systems and is executed by the central processing unit.

4. The computer presents a lesson individually to each student allowing him to proceed at his own speed and governed by the instructional logic designed by the course author. The course itself, in terms of logic and curriculum material exists only in one place in the computer, although from the student's point of view, it appears as if his lesson is unique to him. In other words, the one course must contain all the necessary curriculum and instructional logic to handle all students taking the course. The number of courses available at any one time is a function of the size of the memory systems associated with the computer.

5. It is necessary to have a "time-sharing" computer. Each student is serviced individually,

but the computer attempts to do this with sufficient speed so that each student appears to have sole access to the computer. The speed with which the computer can reply to each individual student (response time) is a function of the number of terminals connected to the computer at the same time, the speed of the computer processing units, and the speed with which the computer can access the curriculum and instructional logic stored in its memory units . . . .

6. A computing system being operated as a CAI system must be capable of recording all actions by the students as they interact with the course. These records are analyzed and form the basis of improving the course content and logic.

7. The following software is required to operate a computer in a CAI mode:

   a. Author Software: a language in which the course author can create his instructional logic and have it present the appropriate curriculum material.

   b. Author Support Software: this software is required in order to make it easy for an author to create his course, e.g., to design different character sets, magnify character sets as might be required for young children, to easily correct and edit textual material for screen display, to trace errors, to collect and analyze performance records, and to obtain documentation.

   c. Operating System: A time-sharing operating system is required. Within this system must be embedded subsystems which make the operating environment of the system an instructional environment. For example, it is necessary to store responses of students as they are made in real-time, have the system sense whether the correct audio and visual materials are available, and to sense a defective terminal which may be located remotely. Subsystems must also be available to allow human intervention in the case of serious programming errors, e.g. to restart a student, back him up, or move him to a help sequence.

   d. A Command Language: required in order that a programmer or author be able to enter and

correct his course material and computer code.

e. Numerical Calculating: the operating system must allow a student access to calculations even from a tutorial lesson; it must be available directly for purposes of calculating. (pp. 13-15)

II. A Review of CAI Languages and Support Systems

A. Definition of Terms

It is assumed that the reader is familiar with most common terms related to computer-assisted instruction. Many of these terms are defined within the text. However, some terms are used in this thesis in a distinct way and are, therefore, defined below.

1. **Building Blocks of CAL:** Godfrey and Sterling (1982) suggest that the basic building blocks of CAL are rules, examples, and questions. Each objective of a CAL lesson is comprised of one or more rules.

   > A rule is any single, testable element of the objective.... [and may be] a rule, a definition, a statement or a practice procedure.... An example is a single expression of any rule. It may include within it expressions of other rules in addition to the rule for which it is designated....A question may be defined as any single query or test situation posed by the computer to the learner, which depends on the learner's mastery of a rule or a set of rules in order to provide a correct answer. (pp. 21-22)

2. **CAI:** Computer-Assisted Instruction or Computer-Aided Instruction - the term used most widely in the United States when referring to instruction administered by a computer. Various modes of instruction are included under CAI: drill and practice, tutorial, simulation, and gaming. In the United Kingdom the preferred term is Computer-Assisted Learning or Computer-Aided Learning (CAL). CAI and CAL are used interchangeably in Canada.

3. **CAL:** Computer-Assisted Learning or Computer-Aided Learning (see CAI)

4. **CAI System User Types:**

   a. Instructors: -those in charge of providing instruction in specific subject matter areas; this may or may not include the actual developer (author) of the instruction.

   b. Proctors: - on site staff responsible for overseeing the interaction between student and machine. They support instructional activities, but are not themselves subject matter experts.

   c. Computer Operators: - those individuals who are responsible for the operation of the hardware and software of the CAL system (this might also involve system programmers/analysts/managers).

   d. Programmers: - those individuals concerned with the instructional programming and probably responsible for making courses from other institutions operational.

   e. Authors: - those individuals who specify the instructional content, logic or strategies of a course.

   f. Students: - those individuals who are the target of the instruction.

A number of important points should be mentioned regarding these different types of users. The distinctions between the first five categories of users will depend upon the size and nature of the CAL system. In a large scale system (e.g., PLATO), it is likely that the instructor (who authors a course) will be different from the person who programs the course and also different from the proctor who is present when the course is used by students. On the other hand, in the case of a school using drill and practice programs (either via a local minicomputer or remotely located large computer), teachers are likely to design, write, and use the programs themselves. Furthermore, in the case of a stand-alone mini- or microcomputer, one person may not only be the instructor, proctor and programmer but also the computer operator too. None-the-less, these five categories of users are distinctive and

> necessary components of a CAL delivery system whether they are fulfilled by a single or different individuals.
> (Hunka et al., 1978, p. 9)

5. **Drill:**

> A drill teaching strategy consists of any combination of rule, example and question. The response to the question is identified as either right or wrong, and some attempt is made to indicate the probable source of error and remedy the misconception. There is no complicated interactive diagnosis of the source of the learner's error. (Godfrey & Sterling, 1982, p. 34)

6. **Inquiry:**

> Inquiry-based CAL is close to the border of information retrieval. The learner indicates which rules he or she wants to see, and is shown only those requested. Inquiry may include examples, but does not include questions based on the rules. Because the basis of this strategy is to allow the learner to select data as he or she wishes, the objective must define the rules which can be accessed by the learner. In many cases, this will be the entire database.
> (Godfrey & Sterling, 1982, p. 42)

7. **Simulation:**

> The main feature of a simulation strategy is a scenario which is displayed on the screen. The scenario is constructed according to a set of rules and examples and is usually designed to represent "real life". The rule and example are combined in ways that force the learner to guess, make assumptions or think in intuitive ways in order to respond to the question. The learner's responses alter the scenario. A simulation may also invite the learner to take control and create the scenario by feeding in the key features (parameters). The scenario which is created will be built according to the rules defined for the objective. (Godfrey & Sterling, 1982, p. 45)

8. **Teaching Strategies:** Godfrey and Sterling (1982) list five basic CAL teaching strategies: Drill, Test, Inquiry, Simulation, and Tutorial (See individual

entries). Some writers include Games as another teaching strategy but Godfrey and Sterling suggest that "each of the five strategies can, with greater or lesser success, become a game." (p. 53)

9. **Test:**

> The question is the only building block used by a test strategy, although responses are evaluated according to a rule or rules. The results may or may not be recorded.... No attempt is made to give assistance or emphasis to areas of weakness. The test may contain questions based on rules that are taught in different objectives in the course. (Godfrey & Sterling, 1982, p. 38)

10. **Tutorial:**

> Tutorial CAL is the most difficult teaching strategy to define because its most important aspect is that it be highly adaptive. It combines all three building blocks, rules, examples and questions, in any way needed to get the job done. Tutorial CAL differs from the other strategies in that in the event of continued learner error the learner goes into an interactive diagnostic sequence designed to determine the source of error. Tutorial CAL may also allow the learner to ask the computer questions about the material presented. Allowing such questions results in a tutorial structure that is highly adaptive and consequently very difficult to construct. (Godfrey & Sterling, 1982, p. 48)

B. A Method for Comparing CAI Languages

To select CAI Systems for review is a difficult task. There are many to chose from though most are available on only a few computer systems. Voyce (1979), during his research to develop a multilingual CAI system, performed an in-depth analysis of five languages - BASIC, CAN-7, COURSEWRITER III, NATAL and TUTOR. All but BASIC are

specialized CAI languages.

Voyce gave the following reasons for selecting these languages:

> 1. BASIC is an example of a general-purpose programming language which has had moderate usage in the CAI environment....
>
> 2. CAN-7 is the CAI language which is almost exclusively used at OISE [Ontario Institute for Studies in Education] and therefore has topical interest.
>
> 3. COURSEWRITER is an example of a very popular programming language which was specifically designed for CAI applications. Version III of COURSEWRITER was selected for this project.
>
> 4. NATAL is the proposed Canadian standard CAI language which contains many features common to other CAI languages.
>
> 5. TUTOR is one of the most frequently used CAI languages.
>
> (p. 36)

The results of his analysis were categorized into a taxonomy of functional properties of CAI languages (p. 66). He found that "although a considerable number of properties are common to all five languages, some properties are either shared by a subset of the languages or are unique to a single language." (p. 41) His purpose was not to compare languages but to describe their functions.

It is the intent in this chapter to make use of Voyce's taxonomy to develop a system for comparing CAI languages. The taxonomy of functional properties was divided into eighteen domains termed CAI language features. Within each domain the functional properties are considered

compensatory. That is, a weakness in one functional property may be compensated for by a strength in another. To compare a set of languages a value is given to each property in a domain that differentiates the languages. The value is in the range of one to three which represents that property's compensatory weight within the domain. When assessing a particular property in one of the selected languages, a value of zero is assigned if the property is not present; or an integer value, up to the maximum set for that property, is assigned which depends on the extent to which that property is implemented in the language.

Once the values for each of the functional properties for all of the languages have been assessed, they are summed for each language within each CAI language feature domain. Thus each language has eighteen scores, one for each feature domain. These scores are used to obtain a profile index in the range zero to ten. This profile index is called a relative prominence index. It is obtained for each feature by dividing a language's score on that feature by the score of the language having the highest score on that feature and multiplying the result by ten.

C. CAI Language Features

Voyce divided the functional properties into six major categories. (p. 39) These major categories are retained for the description of the CAI language feature domains. Four of these categories are further sub-divided to obtain a total

of eighteen feature domains. Each is outlined below. In parentheses after the name of each domain is listed the categories and sub-categories of Voyce's taxonomy (p. 67) contained within the domain. (Note - In Voyce's taxonomy, category 1 is an introduction. The actual taxonomy uses categories 2 to 7. The feature domain categories are renumbered 1 to 6.)

1. Data
    a. Variables (2.1 - 2.2, 2.11) - predefined and user variables, scope of variables, typeless variables
    b. Types and Operations (2.3 - 2.9) - numeric data types and operations, boolean data types and operations, string data types and operations
    c. Pattern Matching (2.10) - elementary patterns, building up subpatterns, continued matching, range of search, run-time parsing of strings, indexing of subpatterns, substring operations
    d. Student's Name, Date, Time (2.12) - special information available to user
2. Data Structures
    a. Organization and Operations (3.1 - 3.4)- random organization, stacks, sets, higher order structures
    b. External Storage (3.5) - sequential files, random-ordinal files, random-keyword files, sequential-random files, operations on files
3. Data Conversion (4.0) - numeric conversion, boolean conversion, string conversion, formatted conversion

4. Program Control

   a. Labels (5.1) - scope of labels, referencing program components

   b. Transfer of Control (5.2) - unconditional transfer, conditional transfer, indexed transfer

   c. Block Instructions (5.3) - looping structures, decision structures

   d. Subprograms (5.4) - functions, subroutines or procedures, invocation of subprograms, nesting, recursion, parameter passing

   e. Stoping and Starting Student Sessions (5.5) - checkpoints, session restarts

   f. Implicit Activity (5.6) - implicit transfers, trapping, asynchronous activity

5. Output to Student

   a. Student Output Devices (6.1) audio devices, slide or microfiche, bell

   b. Terminal Display (6.2.1 - 6.2.3) - dots, vectors, strings, character size, highlighting, fonts, display of variables, display sub-language

   c. The Display Area (6.2.4) - screen size, windows, on other terminals, bounds, text alignment, formatting operations

   d. Additional Display Functions (6.2.5 - 6.2.10) - input display, overprint, displace, display erasing, special display functions, external storage of display information

6. Student Response (7.0) - input devices, time limit, screen area, limit to number of characters, character set, special function keys, responce latency, response position

D. A Comparison of Five CAI Languages

Using the above scheme, it was decided to compare five CAI Languages: CAN-7, COURSEWRITER III, NATAL, TUTOR and PILOT. The first four were selected for much the same reasons that Voyce gave and because data on their functional properties was readily available from Voyce's study. As a fifth language Voyce had selected BASIC. Since BASIC is not a specialized CAI language it was decided to replace it with PILOT, the only CAI language widely supported on todays microcomputers.

An analysis of the functional properties of Apple PILOT (Apple Computer Inc., 1980) using Voyce's taxonomy was carried out. A review of Voyce's analysis of NATAL and TUTOR was also done using the most recent manuals (Honeywell Information Systems, 1981b; Control Data, 1978).

Applying the method for comparing CAI languages outlined in Section B above, scores for each of the eighteen language feature domains for all five languages were obtained. These are shown in Table 1. This table also contains a key to the abbreviations used for the names of the CAI languages. From these scores the relative prominence indices were derived.

Table 1 CAI Language Features: Raw Scores

	N*	T	C	CW	P
1. Data					
1.1 Variables	22	13	4	4	6
1.2 Types and Operations	40	27	22	9	29
1.3 Pattern Matching	28	23	14	11	10
1.4 Student's Name, Date, Time	9	4	2	2	0
2. Data Structures					
2.1 Organization and Operations	15	11	10	7	6
2.2 External Storage	11	3	6	2	5
3. Data Conversion	13	8	6	3	7
4. Program Control					
4.1 Labels	6	3	5	6	4
4.2 Transfer of Control	3	2	3	1	0
4.3 Block Instructions	7	5	0	0	0
4.4 Subprograms	20	11	10	9	6
4.5 Stop/Start Student Sessions	3	2	2	3	0
4.6 Implicit Activity	6	5	2	4	1
5. Output to Student					
5.1 Student Output Devices	6	6	7	7	1
5.2 Terminal Display	11	9	5	1	7
5.3 The Display Area	27	16	7	4	9
5.4 Additional Display Functions	12	12	4	7	5
6. Student Response	10	9	4	10	6

*Abbreviations: N - NATAL, T - TUTOR, C - CAN-7
CW - COURSEWRITER III, P - PILOT

Table 2 Frequency of Obtaining Higher Feature Score

of/over	N	T	C	CW	P	TOTAL
NATAL		16	16	14	18	64
TUTOR	0		13	14	15	42
CAN-7	1	4		9	10	24
CWIII	1	4	5		11	21
PILOT	0	3	7	6		16

Figures 1 to 5 show profiles of relative prominence for each CAI language feature over the five languages. NATAL had the highest score on all features but one. Table 2 lists the number of times each language obtained a higher feature score over each of the other languages. Though each feature is not of equal importance in a CAI language, the table does indicate a ranking of the languages compared. NATAL has the highest rank followed by TUTOR, CAN-7, COURSEWRITER III and PILOT, in that order.

Finally, Figure 6 provides a profile of relative prominence indices for each langage over the eighteen CAI language features. They are displayed on three graphs for clarity.

E. CAI Support Systems

> Although a CAL language provides the basis for the definition of instructional strategies, content, and the sequencing of the interaction of these components with a learner, these components alone are insufficient to provide an effective CAL environment which would stand a chance of competing with, and improving upon traditional modes of instruction. (Hunka et al., 1978, p. 8)
>
> A CAL system consists of a set of interacting subsystems or components based upon a particular author language and operating system which exists to support the activities of the various users of the system. (p. 11)

CAI support subsystems are required to assist in the creation and modification of courseware, the control and monitoring of courseware presentation to students, and the assessment and analysis of the results of student interaction with that courseware. Hunka et al. (1978)

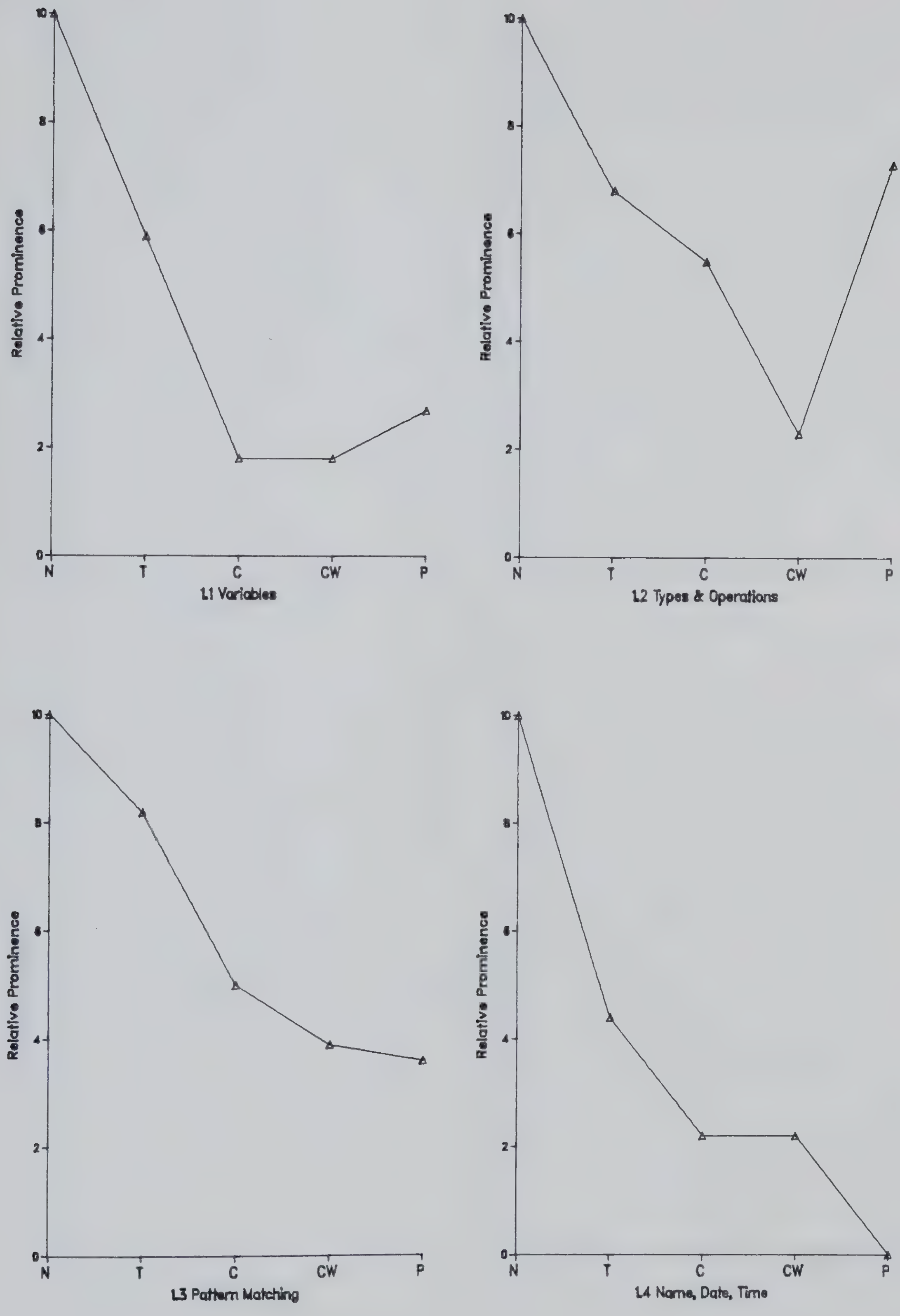


Figure 1 Profiles of 18 CAI Language Features (a)

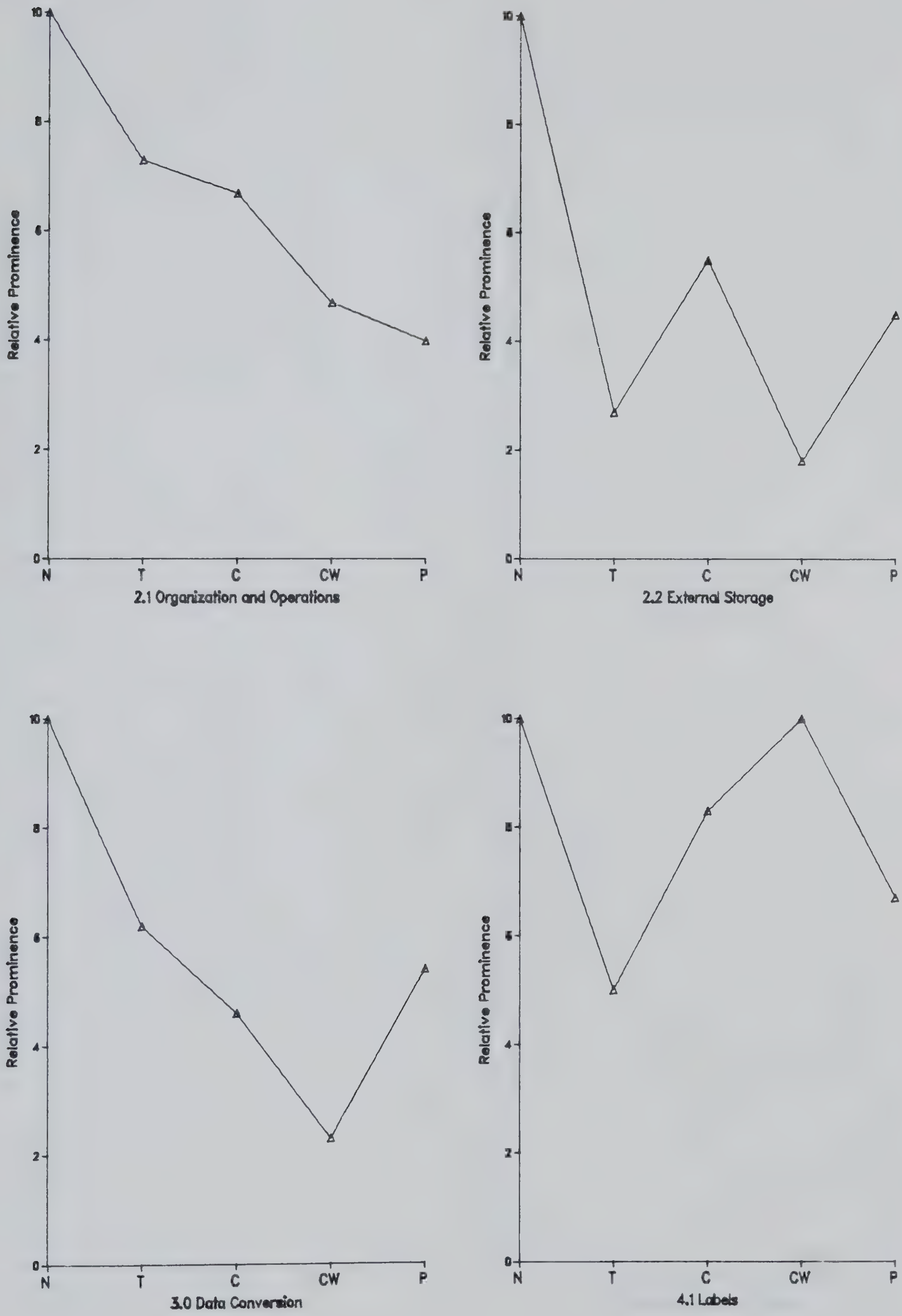


Figure 2 Profiles of 18 CAI Language Features (b)

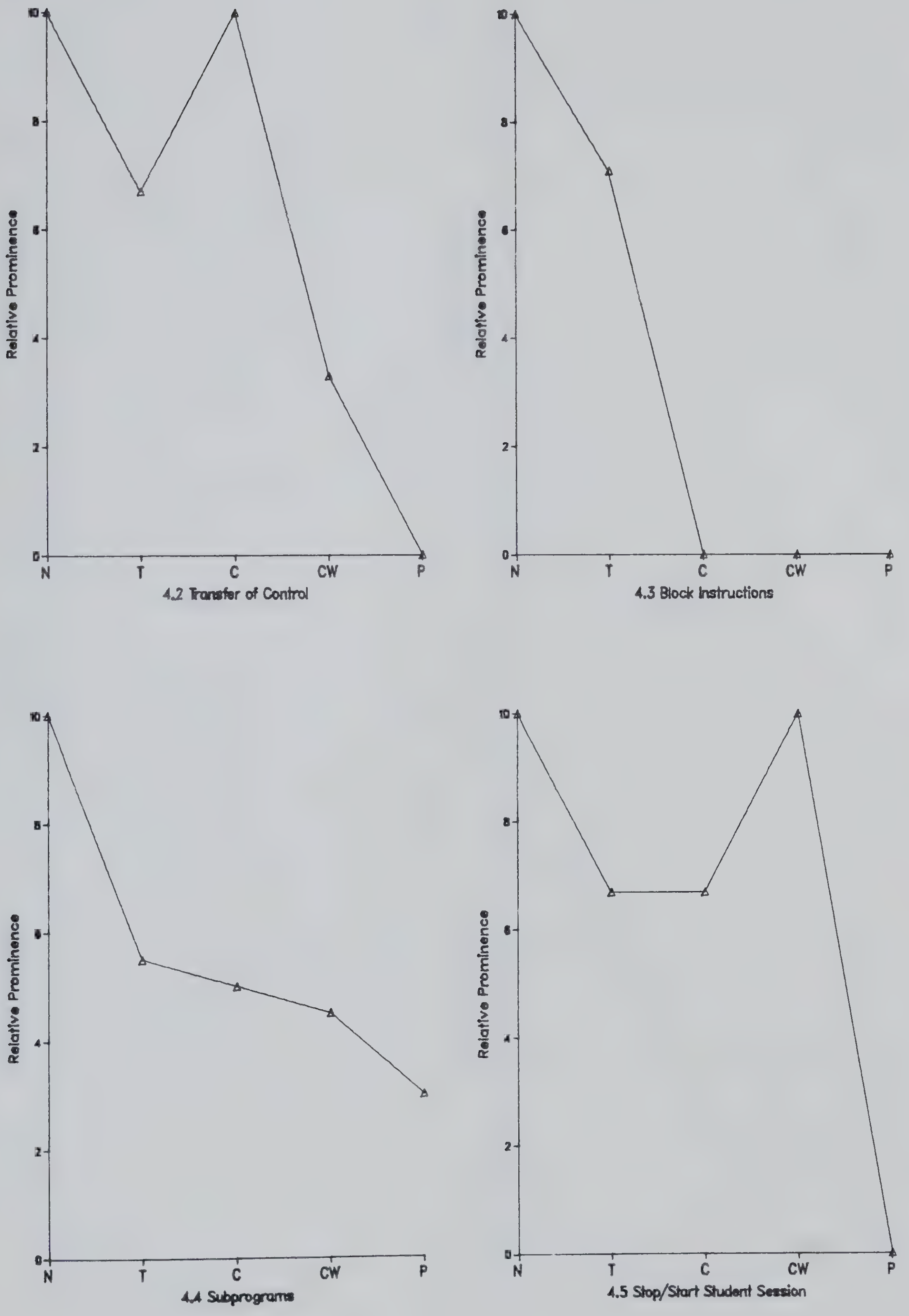


Figure 3 Profiles of 18 CAI Language Features (c)

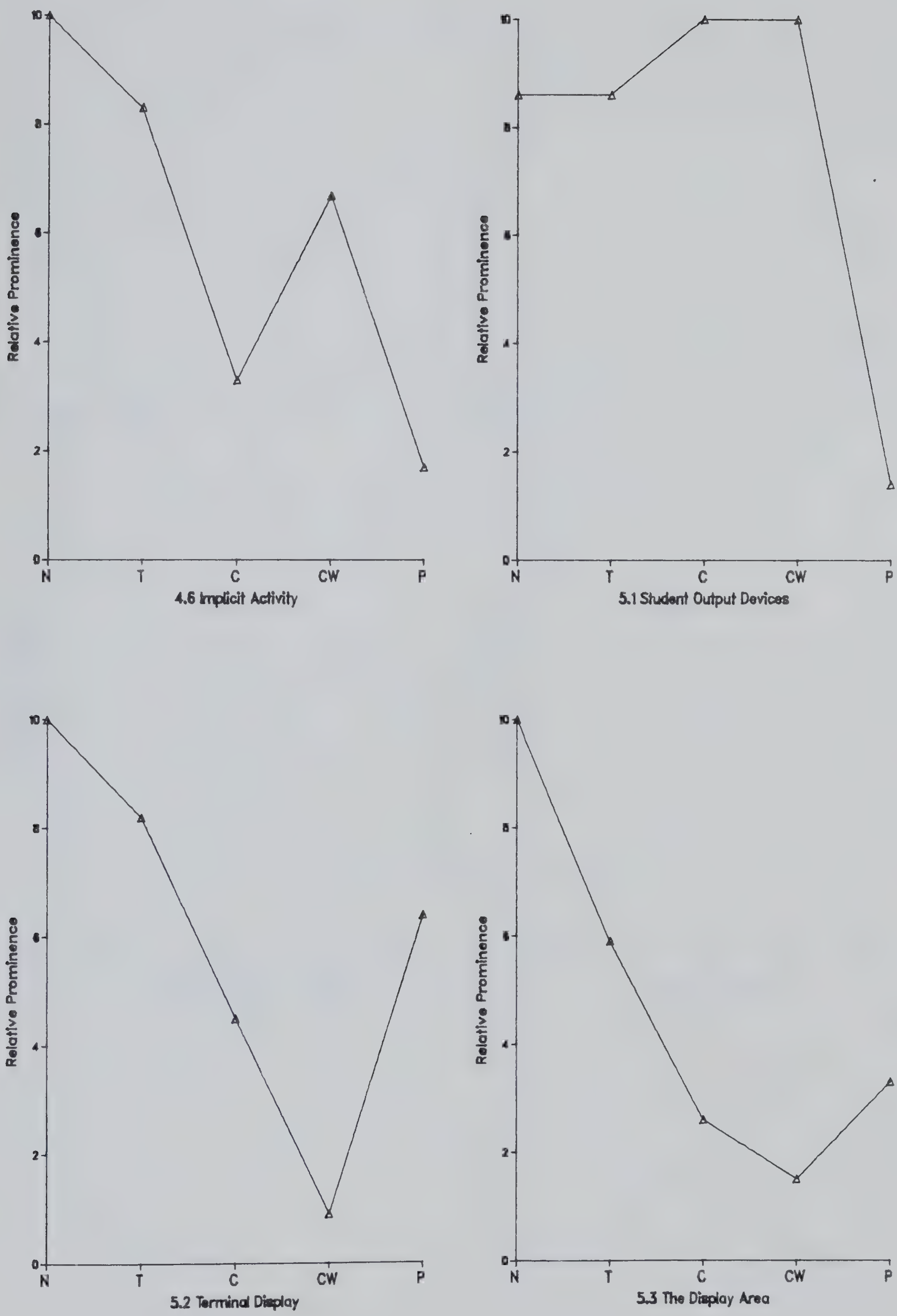
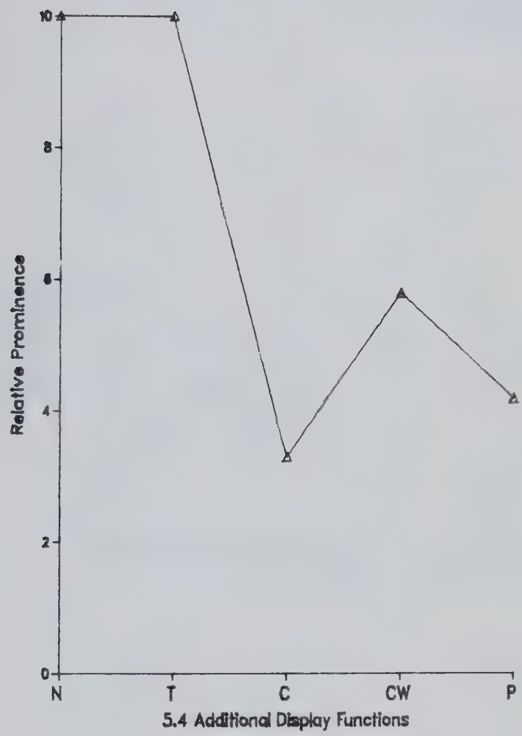


Figure 4 Profiles of 18 CAI Language Features (d)



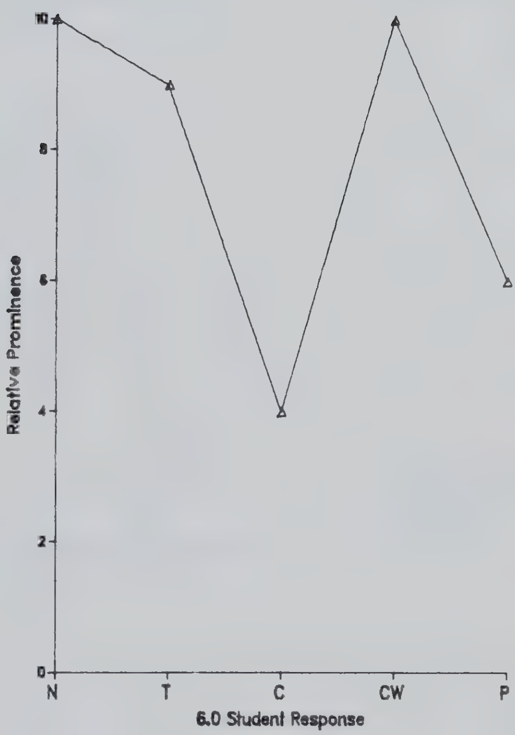


Figure 5 Profiles of 18 CAI Language Features (e)

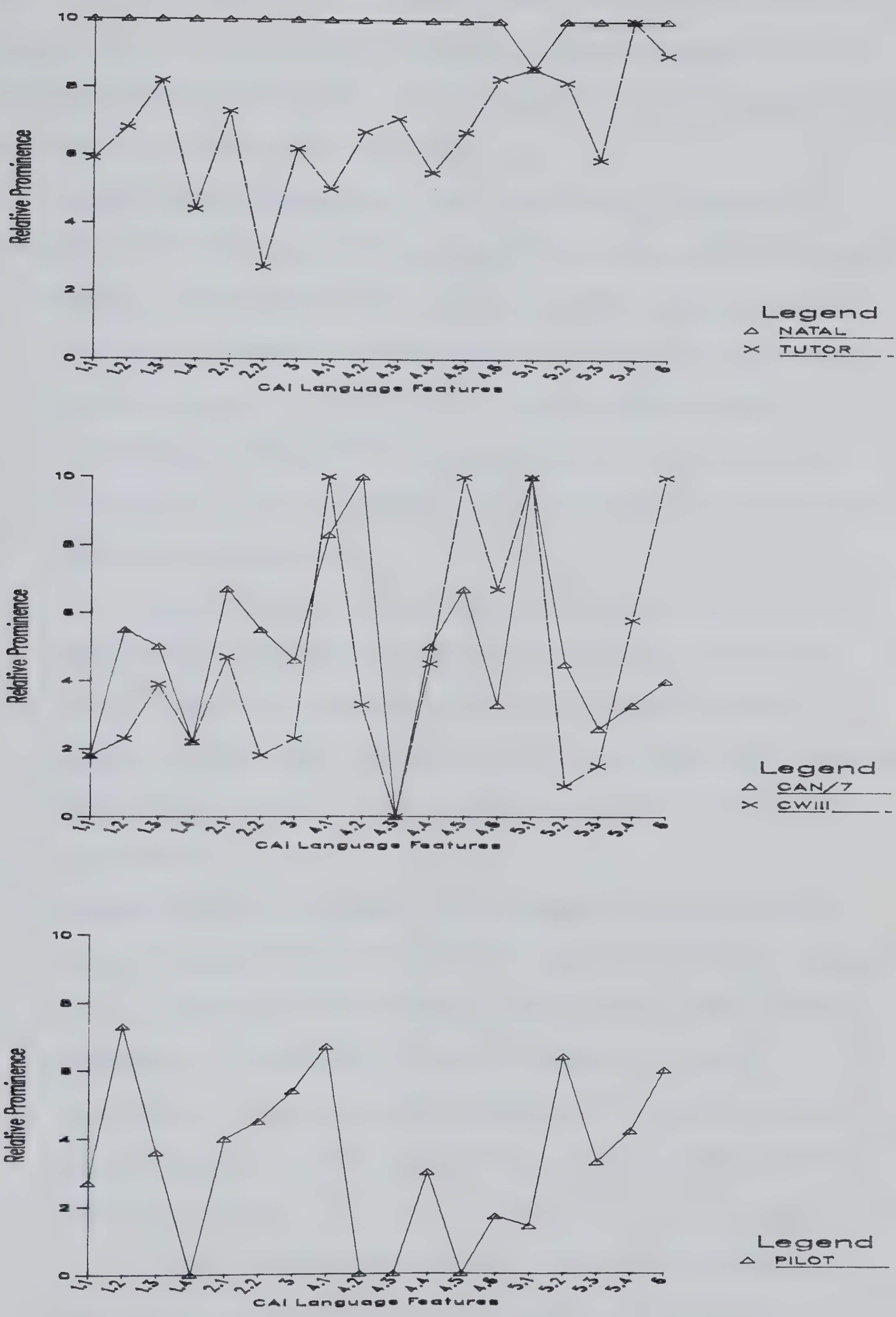


Figure 6 Profiles of Five CAI Languages

defined ten subsystems to support CAI. Some parts of these subsystems are necessary to provide minimal support for an effective CAI environment. Others would be enhancements beyond the minimum requirements.

1. **Registration Subsystem:** This subsystem is needed to establish the identification of all users within the CAI system. Information such as user name, password, course access privileges and subsystem access privileges must be maintained. A method for creating, modifying, displaying, and deleting registration information is part of this subsystem, as are the system's sign-on and sign-off procedures.

   Course registration might be part of this system. Some courses may be available for browsing. In such cases, sign-on registration is not required and a general "demo" user id may be used. At the other extreme some courses may be locked and available to only one programmer. (p. 11)

2. **Communication Subsystem:** This subsystem establishes modes of communications between users. A student comment facility will allow students to interrupt their normal progress in a course to type a comment to their instructor. During the development of a course, these comments may be quite useful to an author when making course revisions.

   On multi-user systems one terminal is usually dedicated to the proctor. If any unusual situation

arises at one of the student terminals, the system could send a message to the proctor's terminal indicating the problem and the terminal location. Such situations as terminal malfunctions and programming errors may be handled this way. Or an author may include a proctor message in the course code to indicate that a student is having unusual difficulty with a particular problem.

An instructor may want to leave a message for his whole class or for a selected student. This can be done with a mailbox facility. When a student signs on, all the messages in his mailbox are displayed before he starts the day's session. In large CAI systems there may also be a facilty that permits real time communications between users. (p. 17)

3. **Documentation Subsystem:** This subsystem provides documentation of courseware and associated subsystems for each user type. For example an instructor needs to know the educational objectives and the associated pre-entry skills, the subject matter content, and the instructional strategies for each major part of a course. A proctor has to have access to all possible proctor messages and related interventions, lists of all questions and answers, and locations of all exams in a course. An operator needs manuals detailing the operation of the hardware and operating system. A programmer must have source code listings to refer to. Finally, a student requires a course outline and

instructions on the use of the terminal and system features, i.e. making comments, using the glossary or calculator, getting hints, or entering learner control mode.

Various types of documentation may be defined. On-line documentation might be part of the course code. The system could interrogate the programmer for this type of documentation during course code programming. Off-line documentation systems using the course code, could generate listings of screen displays, identifier cross references, listings of all questions and anticipated answers and replies by type, and course logic maps. Manuals and guides for all user types are required as is documentation for media use, and font and picture graphics. (p. 19)

4. **Graphics Subsystem:** This subsystem provides facilities for the interactive generation of character fonts not normally supplied on a terminal such as Greek letters or special mathematical symbols, and of graphic drawings for use on either full-graphic or font-graphic terminals. It might also provide for the graphical display of the results of mathematical functions where parameters are supplied by the user.(p. 28)
5. **Examination Subsystem:** This subsystem is concerned with the selection, presentation and scoring of test items in computer administered examinations, and the evaluation and measurement of student performance and instructional

effectiveness. Various exam formats can be supported by CAI Systems. The conventional multiple choice style could be augmented by ranking and confidence weighting techniques. Using the keyword answer analysis features of CAI languages, the short answer format could be employed. With the use of a specialized scoring system, simulations could be used for examination purposes.

Because CAI allows for the individualization of instruction, students will often reach particular examinations at different times. This presents a security problem for those examinations. The system can assist here by presenting the test items in a randomized order or by selecting test items from a stratified pool of items. Tailored testing, where the next question given a student is contingent upon the response to a previous question, could be supported; as could the generation of random numbers, within intervals specified by the author, to be used as problem parameters in math test items.

Different presentation and feedback strategies may be available to the instructor or presented as options to the students. For example, the student might be allowed to preview all questions before answering, select the order of answering, or change answers to previous questions. The amount and timing of feedback on responses could be altered. Finally, the system must route examination results to the appropriate people.

(p. 32)

6. **Student Record Subsystem:** This subsystem collects and analyzes student performance records for the purpose of monitoring the progress of a single student or a group of students, optimizing a CAI course, and identifying "bugs" in a CAI course. Performance records must contain enough information for various analyses to be carried out. The system will define a basic set of variables to be saved for each type of performance record, but an author may also define additional variables to be saved for specialized analysis. A performance record should be saved each time a question is asked, an examination is completed and scored, an error condition is encountered, or a student signs on or off, passes a restart point or performs some predefined "special" action. A set of analysis routines should provide cumulative results of a group of students on a particular question, and the cumulative performance of all students within a section of a course or of one student for all sections of a course. A more advanced feature would be the capability to retrace the exact path of a student through a section of a course. (p. 39)
7. **Exception Handling Subsystem:** When an error condition arises during the execution of a course, this subsystem should route an error message to the individual who can correct it: operator, proctor, instructor, programmer, author or student. If sent to the student the error

message must be meaningful and specify corrective action. If an error condition causes execution to halt, a meaningful display should be presented to the student until the problem is fixed or the student is rerouted. To ensure that proper corrective action is taken all error conditions should be recorded in an appropriate system file as well as sent to a printer. (p. 45)

8. **System Monitoring Subsystem:** This subsystem provides routines for the general monitoring of system functions and records. The following are some of the facilities that this subsystem could provide:
   a. brief on-line reports indicating where each student is relative to the course code and each other, and his cummulative signon time and last signon date
   b. interactive queries for selected information in students' restart and status records
   c. identification of current execution location for debugging purposes
   d. on-line diagnostic procedures to flag terminal malfunctions
   e. access to operational status of each version of every module in a course
   f. access to current user status
   g. accounting information on system use.

(p. 46)

9. **Execution Subsystem:** During execution of a course, an instructor may allow a student to make use of a review

mode. In this mode a student may suspend the normal flow of the course to back up one or more "frames", to back up to some logical point and then retrace his path with the system supplying his previous responses, or to branch to another section of the course. Various facilities could also be allowed for the student. He may have access to a course glossary, help or hint messages, a calculator mode, or a plotting procedure. If audio messages or pictures are presented, the system should allow the student the ability to request that the last one presented be repeated. A course map could be made available to the student that would show him where he is in the course and which sections of the course have been completed. It might also be possible for a student to have access to certain information in his personal record area or summaries in the group record area.

Certain execution time facilities can also be made available to proctors. There may be a need for a proctor to intervene in the normal operation of the system to redirect a student to a different section of the course or to suspend course execution. The proctor should also be able to change variables in the student record area, set the system latency default, and monitor a particular student's progress in parallel terminal mode.

When an author or instructor is reviewing a course, he should be able to use pseudo answers like "c" or "w" for correct or wrong answers and be shown the expected

response and associated reprise display. (p. 52)

10. **Course Entry, Modification and Testing Subsystem:** This is one of the most important subsystems because it is the first encountered by beginning programmers and is used by personnel with the widest range of experience.

Course entry may be made "on-line" or "off-line." On some of the larger CAI systems a course author may pass the course design to a programmer for encoding. However, in most systems the author programs his own course code and is usually not a professional programmer. The system should provide the author /programmer with aids to assist with course entry and modification. Input specification forms are often provided as off-line support. These may include conventional coding forms, screen display forms and "fill-in-the-blank" instructional model forms. These can be used by data-entry operators to enter course content directly. Once a programmer has corrected all syntactic errors, the author should be provided with screen by screen listings and a course map listing to assist in checking course logic.

During on-line code entry each statement should be checked when it is entered for syntactic correctness, and errors marked for immediate correction. Code modification could be performed by a powerful but simple text editor with both line and screen editing modes. The author should be able to set up screen displays using a

full screen editor and save these displays by logical names for recall by the course code.

Pre-defined instructional models may be available. They can greatly assist beginning authors by allowing them experience with well defined models. They do, however, limit the options available to the author.

Procedures for testing and debugging programming and instructional logic should be provided. This might include a driver program that simulates a student taking a particular course module. (p. 59)

III. Requirements for a Microcomputer Based CAI System

A. Requirements for Courseware Development

The development of CAI courseware not only requires knowledge of subject content, but also knowledge of instructional design and skill in computer programming. Taking a rather broad view of instructional design, it might include the appropriate recognition of any of the following factors:

- curriculum design and subject matter sequencing
- graphic design
- recognition of a student's level of development
- visual perception
- motivation
- diagnosing and remediation
- testing
- learning style.

Figure 7 illustrates the intersection of the three major area competencies. The range of knowledge or skill shown is from adequate (minimum for good courseware development) to superior. It is unlikely that a person with superior subject content knowledge would also be superior in the other two areas (vertex A). To successfully develop superior CAI courseware, a team of specialists in subject content, instructional design, and computer programming is required. It would not be unusual, however, to expect that individuals

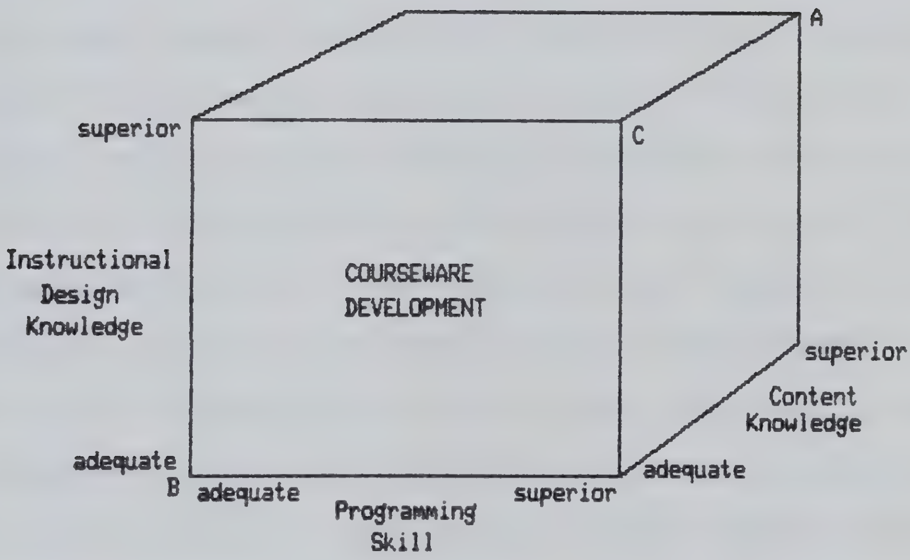


Figure 7 Courseware Development Competencies

could gain at least adequate knowledge and skill in all three areas (vertex B) and produce adequate courseware if given adequate tools.

There will be others having the requisite subject content knowledge and the desire to develop courseware, while having less than adequate knowledge and skill in the other two areas. If a CAI system can separate instructional strategy from subject content, a team of programmers and instructional design specialists (vertex C) could develop sets of CAI strategies that content specialists could utilize in preparing courseware. In the drill and practice mode of CAI, this might take the form of providing parameters to math drills or spelling word lists for use in a 'Hangman' game. In the tutorial mode, the subject specialist, if provided with the proper programming tools, might develop elaborate instructional interactions that could include animated displays and complex answer analysis.

The CAI system must attempt to supply the needs of the courseware designer who is attempting to produce the most effective instruction possible over a wide range of content and instructional strategies. Some of these needs are:

- easy control of branching and return, e.g. going to a glossary or help sequence at the student's request
- score keeping and internal monitoring
- monitoring of student progress
- access to numerical calculating power
- flexible input

- good answer analysis
- easy creation of graphics and animation
- easy procedures to review and test courseware
- tailoring courseware to individual student needs, e.g. use of external parameters to adjust pathway through a course.

With reference to computer languages and support systems, the term "user friendly" has come into vogue. This usually means that the languages and systems are designed to be supportive to both the beginning and experienced programmer. This is especially important in CAI since many courseware designers may not have had a great amount of programming experience. A CAI language and system should be easy to learn and use. Courseware entry, editing, and testing should be straight forward with good error checking and, perhaps, interrogative coding in which the system queries the programmer for each program element, accepting only legitimate input. The language should provide a rich variety of intrinsic functions and operations, and, for the experienced designer or researcher, should allow for easy extension for specialized needs.

B. NATAL-74: Canada's NATional Authoring Language

Romaniuk (1970) developed the CAI language VAULT which provided for the separation of instructional design and subject content. The instructional design specialist programmed in the VAULT logic division while the content

specialist programmed in the simpler data division. The VAULT compiler combined the two divisions to produce executable Coursewriter II code.

A more recent development of a CAI language that provides for this separation of strategy and content is NATAL-74. As was shown in Section D of Chapter II, NATAL also had the highest score on all CAI language features but one when compared with four other prominent CAI languages. For these reasons NATAL was selected as the model upon which to base the design of a microcomputer based CAI system.

"NATAL-74 was commissioned by the National Research Council in 1972 and designed by the IBM Canada Lab to meet specifications produced by a panel of CAI users from across Canada." (Westrom, 1976, p. 1) It is really three languages in one: a procedure and function language, an instructional unit language, and a display sub-language. The latter is a sub-language of the instructional unit language. Figures 8 and 9 illustrate the relationships among procedures, functions, and units.

The instructional strategy of a course is written in a procedure language which bears a resemblance to PL\1 or ALGOL. Support routines are also written in this language. These include General Functions, which serve the same purpose as functions in any computer language; Edit Functions, used to edit student responses before answer analysis processing; Graphic Functions, to provide author defined graphic displays; and Font Blocks, for the

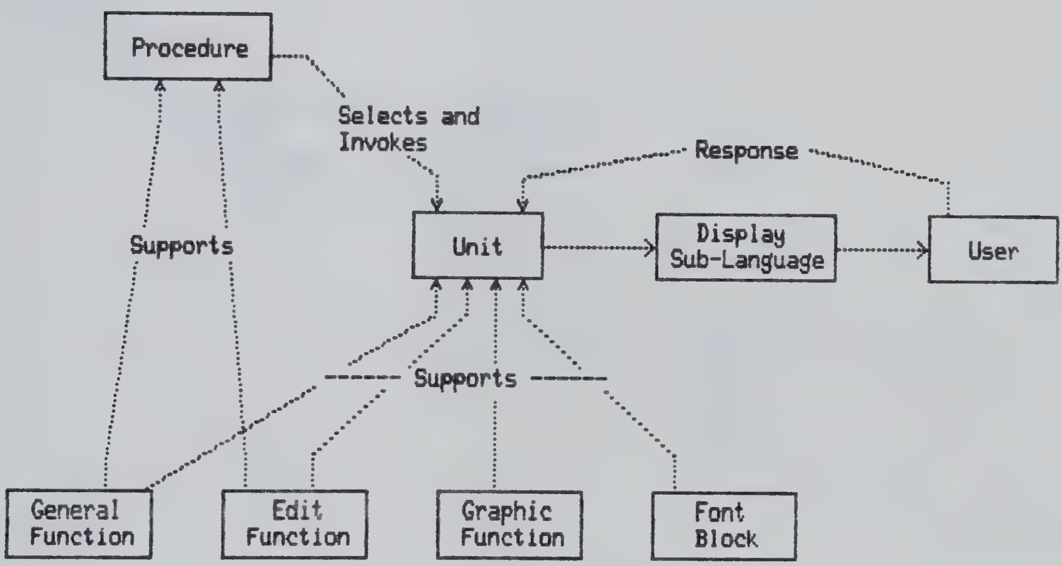


Figure 8 Unit, Procedure, and Function Relationships

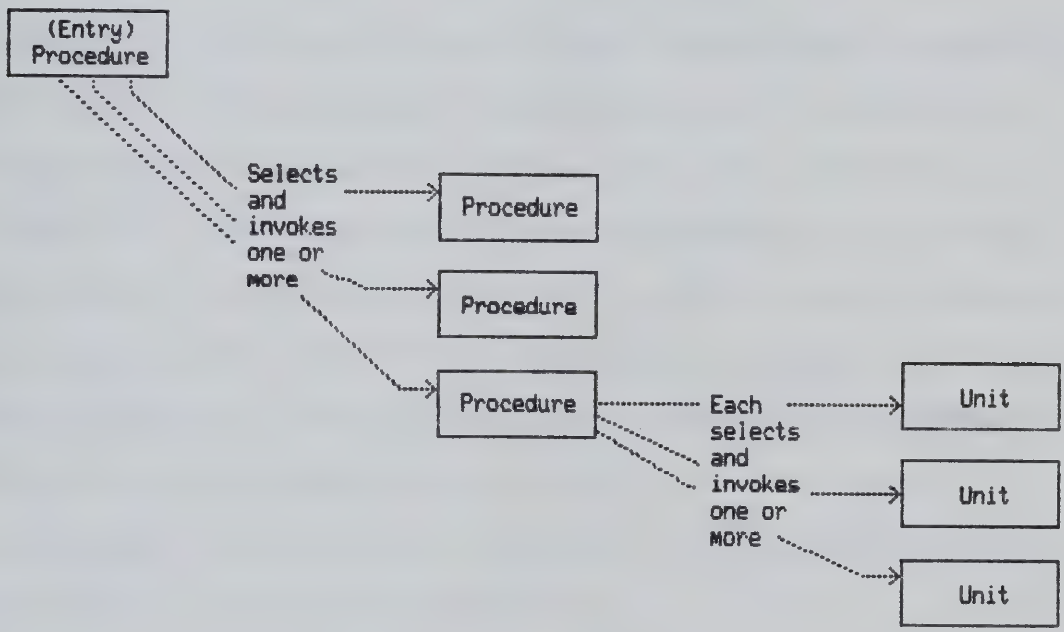


Figure 9 A Course Hierarchy

specification of additional character fonts. The language also provides a set of intrinsic general, edit, and graphic functions.

Content presentation and instructional interaction with the student are provided through a special routine called the Instructional Unit, or just Unit. It has its own language and intrinsic branching mechanism. The third language is called the Display Sub-language which controls the display of characters and graphics on the student's output device in a manner similar to a word processor. It allows for the formatting of the same material on devices with differing characteristics. The Display Sub-language is only used within the Unit. A commercial version of NATAL-74, termed NATAL-II, has recently been developed by Honeywell Information Systems Canada. This will be the version referenced throughout the remainder of this thesis. (see Honeywell, 1981a, 1981b, 1981c)

C. Selection of a System Development Language

NATAL-II is too large and complex a language to be easily implemented on the microcomputers most often found in schools today. However, the design philosophy of NATAL, separating strategy and content, could be incorporated into an extended version of a language already available on these same microcomputers. That is the approach to be taken in this thesis project. The name selected for this CAI development system is CASTLE, an acronym for

*C*omputer-*A*ssisted *S*tudent *T*utorial *L*earning *E*nvironment.

Which language should be selected as the base language on which to build this CAI system? Since, as a minimum, it must serve the same function as the procedure language of NATAL-II, it should be a structured language. For this reason BASIC, the most common language on microcomputers, must be rejected. BASIC has limited structured control statements, does not have named procedures and functions, does not permit parameter passing or local variables, and variable names are significant only to the first two characters. The main advantage of BASIC is that it is easy to program and debug because of its interactive interpreter.

Pascal might be a likely candidate. It is a structured language with all the characteristics that BASIC lacks. But it has poor string handling capabilities needed in any CAI environment, and programs are more difficult to create and debug since it is a compiled language. (UCSD Pascal does have excellent string handling capabilities, but is encased in a complex environment. The Apple II version requires two manuals of over 500 pages just to explain how the system works and how this version of Pascal differs from the standard.)

The language selected as the base language for the CASTLE CAI development system is COMAL-80. Developed originally in Europe as a structured replacement for BASIC, it has attracted considerable attention on this side of the Atlantic after Commodore International placed in the public

domain in May 1981 a COMAL-80 interactive editor and interpreter. COMAL (COMmon Algorithmic Language) has been referred to as the language that combines the best of BASIC with the best of Pascal. It contains all the structured control statements of Pascal; permits long name identifiers for labels, variables, and procedure and function names; allows parameter passing as call by value or call by reference; and supports local and global variables. In addition, it has excellent file handling characteristics supporting read, write and append to sequential files as well as random access files. The interactive visual editor provides for modular program development, automatic syntax checking on input, and automatic 'pretty print' listings to the screen or the printer. One of the main reasons for the selection of COMAL-80 is its string handling capabilities, superior to both BASIC and Pascal. These include selection and assignment of substrings, and a string inclusion function that returns the index of a substring in a string.

The CASTLE system, to some extent, will be developed as a NATAL look-alike. The purpose of this approach is to facilitate programmer portability between the two systems, and, perhaps, courseware transferability.

D. The CASTLE Instructional Environment

Any computer language may be used to implement CAI. CAI lessons have been written in BASIC and delivered to the student via teletype terminals. The facilities of the

language and associated hardware may help or hinder the realization of any particular CAI teaching strategy. The extent to which a teaching strategy may be effected depends on the instructional environment supported by the software and hardware utilized.

In this context an instructional environment may be classed as verbal, numeric, spacial, auditory, and/or kinesthetic/motor. A verbal environment requires that the system can present the full range of alphanumeric and punctuation characters in whatever font is required for a given lesson. The student should be able to respond in the same character set and to alter his response before passing it to the computer for analysis. The CAI language should support statements for editing the student's input and have the capability to do complex analysis of the student's response, including checks for misspelled words, by searching for the presence or absence of key words or phrases in or out of a prescribed order. The CASTLE design fully supports the verbal environment. The ability to match phonetic targets is not implemented at this time, nor is the ability to design and change character fonts.

A numeric environment requires that the system can present the full range of mathematical symbols and structures. The student should be able to respond with this same set of mathematical symbols and should be able to access the calculating power of the computer. The CAI language should be able to scan input for acceptable

mathematical symbols or numerals, to analyze input for numeric equality within a specified tolerance, and to compare algebraic input for equivalence to an algebraic target. The language should support both scalar and matrix algebra. The CASTLE design partially supports the numeric environment since only scalar algebra is included. Just the mathematical symbols contained in the standard ASCII character set are implemented, and the algebraic comparison function is not implemented.

A spacial environment requires that the system can present high resolution colour graphics representative of both 2-D and 3-D figures. Input devices for this environment might be a light pen or touch screen, a graphics tablet, or a "mouse". Other visual presentation systems, such as a slide projector, microfiche or a video disk under the control of the CAI software, might be considered part of this environment. The CASTLE design does not support the spacial environment but could be easily extended to include this environment.

An auditory environment requires that the system be able to generate voice information, music and other sound effects, and to present pre-recorded sound. Auditory input and voice recognition are still mostly experimental. A kinesthetic/motor environment has been associated with complex flight simulators used by commercial airlines and the military. The CASTLE design does not support either the auditory or the kinesthetic/motor environment.

IV. Design Specifications for the CASTLE Language

A. Overview

A CASTLE course is composed of one or more chapters. A chapter, in turn, is composed of one or more lessons (see Figure 10). A chapter should be considered as a major division of a course that can stand on its own, and must have a unique name within a course. An author may define a default order for the chapters in a course but it is the responsibility of each instructor to select and arrange the order of the chapters. This may be done for a class and can be altered for individual students. An instructor has the option of selecting additional chapters from an external source such as another course or a system library.

The order of lessons in a chapter is, however, primarily, under the control of the author. The author defines a default order for lessons within a chapter, but from within any lesson, there may be any number of exits to other lessons. Thus the author may individualize each student's pathway through a chapter (see Figure 11).

The lesson is the basic control module in a CASTLE course. It must have a unique name within a chapter. Lessons are written in the CASTLE procedure language. Each lesson begins with a lesson entry procedure named 'lesson'. A procedure is a computer routine, or sub-program, that controls the actions of the computer. In CASTLE, procedures

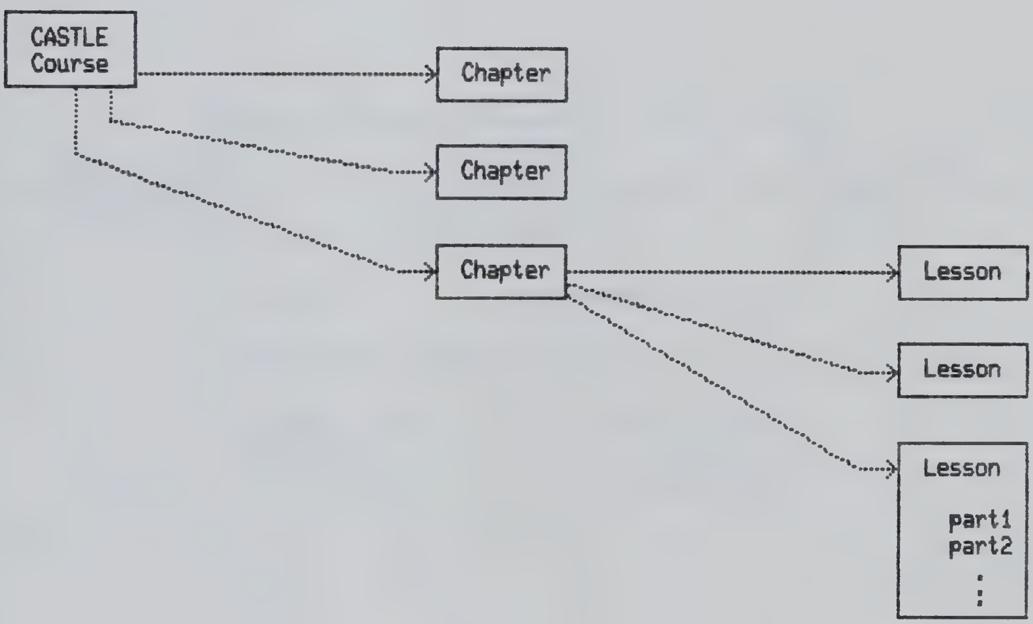


Figure 10 Hierarchical Control of a CASTLE Course

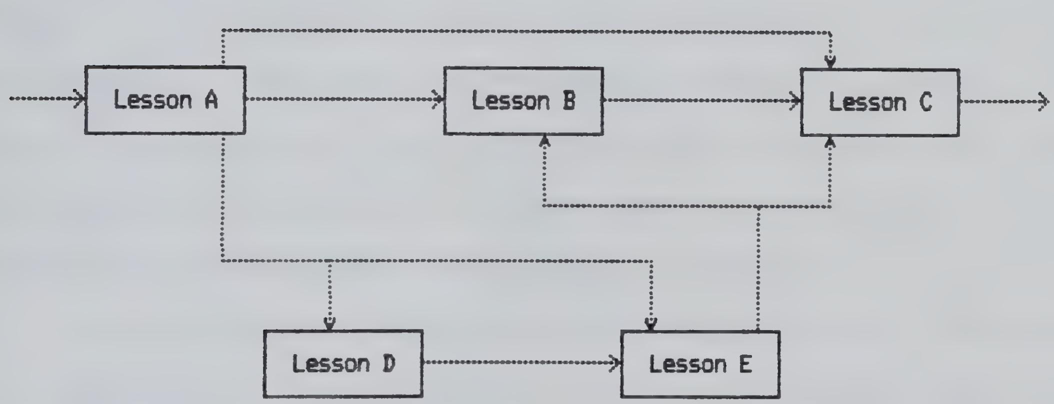


Figure 11 Illustrative Example of Lesson Selection Within a Chapter

are used to select instructional units, gather data on computer interactions with the student, and make decisions on choosing the student's pathway through a course.

The author may divide a lesson into parts. A part might be associated with a single objective or sub-objective of a lesson. Upon re-entering a course the student is automatically routed to the beginning of that part of the lesson which was being executed just prior to sign off. Each part of a lesson has a unique number and name within the lesson, and begins with a part entry procedure having the part's name. Parts would normally be executed in numerical order but the author has complete freedom to call parts in any order. For example, a student having difficulty in part 4 may be re-directed by the author to part 2.

Within lessons, procedures may be defined by the author for the general control of the lesson. A procedure may be called by another procedure and is then subordinate to the calling procedure (see Figure 12).

Instructional transactions are carried out by unit routines. All interaction with the student is handled by these units. The unit is used to present information to the student, to pose questions and to analyze the student's response. System variables are used to pass information about the transaction back to the calling routine. Units are called from procedures in the lesson modules. Units may have names that are known to any course in the CASTLE system or their names may be local to a course, a chapter or a lesson.

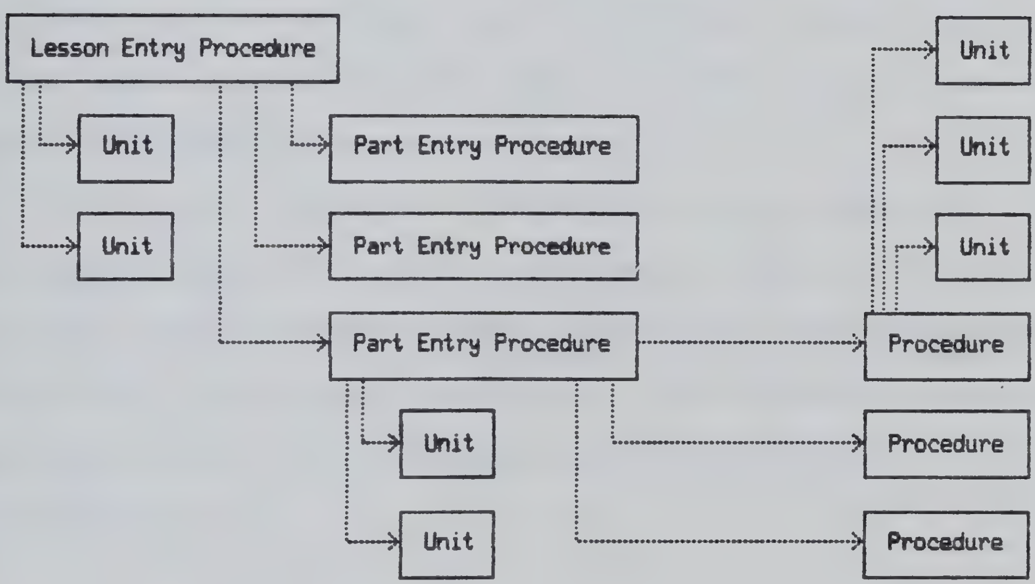


Figure 12 Procedures and Units in a Lesson

The presentation of information is handled by a special display sub-language that formats text and graphics on the student's display devices in a manner similar to a word processor.

Routines that support both instructional transactions and control procedures are called functions. These are classed as general functions, edit functions, comparison functions, and graphics functions. The CASTLE system provides a number of functions in all four classes but the author may define unique function's in each class using the CASTLE procedure language.

The CASTLE Language is composed of three separate language systems that share system data. The system data are of three classes: system files, system registers, and system variables. The language systems are the procedure language, the instructional unit language, and the display sub-language.

B. System Data

System Files

There are five types of system files: lesson module files, unit parse table files, unit string table files, display files, and window files. The lesson module file type contains CASTLE system routines and author defined procedures and functions. The two unit file types contain

control and display data for the unit language interpreter. The remaining two file types are referenced by the display sub-language interpreter. Display files contain data to be formatted by the display sub-language interpreter. Window files are preformatted data to be displayed in an author-defined window on the student's display screen. These files are referenced by file identifiers which may have up to twelve alphanumeric and/or symbol characters except the quotation mark, semicolon, asterisk, question mark and comma, but may include the space character. The first character must be alphanumeric. A file identifier may be known to the entire CASTLE system or may be local to a particular course, chapter, or lesson. When referenced from within a course, a CASTLE system file identifier is prefixed with an '&'; a current course file identifier, with a '#'; a current chapter file identifier, with a '%'; and a current lesson file identifier, with no prefix.

Chapter and course names must have the same type of twelve character identifier. Unit and lesson identifiers may also have numeric indices. They appear after the identifier, which is then restricted to ten characters, and range from '00' to '99'.

System Registers

For each student there is a set of local and global registers. In each set there are 100 numeric registers numbered 0 to 99, and 21 string buffer registers numbered 0

to 20. The numeric registers may hold integer or real numbers. The string buffer registers may contain strings of up to 80 characters. The local register set is local to each lesson module. They are referenced as system variables c(0) to c(99) and b$(0) to b$(20). The global register set is global to the entire course. They are referenced as system variables c'(0) to c'(99) and b'$(0) to b'$(20).

System Variables

All system variables are local to each lesson module. They are known only to the procedures and functions in that lesson module and to any units called from these procedures. They will be discussed in the section on system variables below.

C. The Procedure Language

The CASTLE procedure language is an extended version of CBM COMAL-80 revision 01.02. Reference will be made to various elements and features of COMAL in the following descriptions. These will not necessarily be complete. For a complete description of COMAL see Lindsay (1983).

COMAL routines are written as a series of statements. Each statement is preceded by a line number in the range 0 to 9999. Generally there is only one statement per line. The line numbers are only used by the COMAL editor and are not part of the routine statement. COMAL has three types of routines: programs, procedures, and functions. CASTLE

authors will only write procedures and funtions. The CASTLE system routines are written as COMAL programs, procedures, and functions.

When describing the syntax of CASTLE statements the following conventions will be used:

1. Items in UPPER CASE are keywords and must be typed as shown, unshifted.
2. Items in lower case and enclosed in angle brackets <> are supplied by the user. The angle brackets are not typed.
3. Items enclosed in square brackets [] are optional. If used the square brackets are not typed. If the item enclosed in square brackets is followed by an ellipsis (...), it may be repeated as often as needed.
4. All other punctuation should be typed as shown, including parentheses ().

An <identifier> is a name of up to 78 characters, each one significant. These characters may be unshifted alphabetic, numeric, apostrophe ('), backslash, square brackets ([or]), and left arrow. The first character must be alphabetic. An <identifier> may be used as a variable name, line label, function name, procedure name, or array name.

Data Types

CASTLE supports integer, real, and string data types. Integer literals are whole numbers in the range -32768 to +32767. Real literals may be expressed as integers, as

decimals, or in E-notation. The E-notation exponent must be in the range -38 to +37. Up to 9 significant digits can be represented. A string literal is 0 or more characters enclosed in quotation marks. Within a string, a quotation mark is represented by two consecutive quotation marks.

Real and integer variables do not have to be explicitly declared. A real variable is represented by <identifier>. An integer variable is represented by <identifier>#. A string variable is represented by <identifier>$. String variables must be declared by a COMAL DIM statement as having a maximum length of from 1 to 32767 characters (Lindsay, 1983, p. 34). Arrays of all three types may be declared by a DIM statement and may be multi-dimensional (pp. 35-38).

Functions that return integer or real results may be defined. The function name has the same form as a real or integer variable. String functions cannot be defined but can be simulated by a procedure with a call-by-reference string parameter.

Switch or boolean is not a separate type but is represented by the numeric type. A value of zero represents FALSE and any other value represents TRUE. The switch literals TRUE and FALSE are supported. Since file names are strings, file variables are simply string variables.

String, numeric and boolean expressions are supported by CASTLE. The operators are: ( ) | * / DIV MOD + - = <> < > <= >= IN NOT AND OR. See Lindsay (1983, pp. xxi and 288) for details.

Routines

A course author may define five types of routines: the procedure, the general function, the comparison function, the edit function, and the graphic function. Each routine has a header statement, a routine body, and an end statement.

The form of the procedure and general function is similar (Lindsay, 1983, pp. 231-235) and given below:

```
PROC <procedure name>[(<formal parameter list>)] [CLOSED]
  [IMPORT <identifier list>]...
  <statements>
ENDPROC <procedure name>

FUNC <function name>[(<formal parameter list>)] [CLOSED]
  [IMPORT <identifier list>]...
  <statements>
ENDFUNC <function name>
```

A procedure is called by simply using it's name, for example,

```
<procedure name>[(<actual parameter list>)]
```

The same is true for a general function except that it's name must be used in an expression, as for example,

```
<variable> := <function name>[(<actual parameter list>)]
```

Formal parameters may be either called by value or called by reference. A reference parameter name is preceded

by the keyword REF. A value actual parameter is an expression of the same type as the corresponding formal parameter. A reference actual parameter is a variable of the same type as the corresponding formal parameter. Array names may also be passed to REF parameters. The formal parameter then becomes a nickname for the actual parameter. Thus values may be passed in and/or out of procedures and functions through parameters. Parameters in a list are separated by commas. Parameter names are local to their routine.

The use of the CLOSED key word makes all identifiers within the routine local and locks out all global identifiers. Global identifiers may be imported into a closed routine by use of the IMPORT statement.

A RETURN statement may be used anywhere within a procedure and when executed causes an immediate return to the calling routine. If there is no RETURN statement then a return to the calling routine occurs when the ENDPROC statement is reached.

The form of the RETURN statement in a general function is as follows:

RETURN <non-string expression>

When executed the expression is evaluated and its value returned as the function's value in the calling expression. Execution of the function immediately terminates.

There may be any number of RETURN statements in a routine. A function must be exited through a RETURN

statement.

Routines may be defined within routines and may be called recursively. The unit interpreter cannot be called from within a closed procedure.

Comparison functions are boolean functions used to compare a target response to a student's response in the system variable redited$ or the system vector nedited. NATAL-II defines six system supplied comparison functions: ca - compare algebraic, cc - compare character, cg - compare graphic, ck - compare keyletter, cn - compare numeric, and cp - compare phonetic (Honeywell, 1981b, pp. 85-89). CASTLE supplies three of these: cc, ck, and cn. They are described in the section 'System Functions' below. A comparison function must return a value of TRUE or FALSE and may be used in a boolean expression. They are called in a unit categorization statement.

An author may define up to ten comparison functions in one lesson, having the names ucf0 to ucf9. ucf is a mnemonic name for 'user comparison function'. The syntax for the ucf header statement is:

```
FUNC UCFn(<string variable>, REF <numeric vector name>())
```

A target string might be passed to the first parameter. The second parameter is a vector with indices 0 to 10. Index zero contains a count of the number of remaining indices referencing usable data.

Edit functions are actually procedures used to edit a student's response in the system variable redited$ or the system vector nedited. NATAL-II defines eight system supplied edit functions (Honeywell, 1981b, pp. 93-94). CASTLE provides all of these. They are described in the section 'System Functions' below. Edit function names may be assigned as a list to the system variable sedit$, separated by commas and enclosed in quotation marks. The edit functions assigned to sedit$ are executed in order after a student response is received in the execution of a unit. Edit functions may also appear in unit edit statements.

An author may define up to ten edit functions in one lesson, having the names uef0 to uef9. uef is a mnemonic name for 'user edit function'. The syntax for the uef header statement is:

```
PROC UEFn
```

Graphic functions in NATAL-II, both system provided and user defined, are called by the '&G' command in the display sub-language to create graphic displays on the student's display screen. This feature is not fully implemented in CASTLE. User defined graphic functions when called by the '&G' command in CASTLE allow the author to take over the control of the student's display from the display sub-language. The statements in the graphic function may incorporate any of COMAL's display control features. The few

system supplied graphic functions in CASTLE are described in the section 'System Functions' below. These may not be called by the '&G' command but may be used in a user defined graphic function.

An author may define up to ten graphic functions in one lesson, having names ugf0 to ugf9. ugf is a mnemonic name for 'user graphic function'. The syntax for the ugf header statement is:

```
PROC UGFn
```

Control Structures

The power of structured languages lies in their modular structures and their control structures. The modular structures of CASTLE were discussed in the previous section. The control structures of CASTLE are those of COMAL (Lindsay, 1983, pp. 224-231).

There are two types of control structures: decision structures and looping structures. The decision structures are the IF structure and the CASE structure. The looping structures are the FOR structure, the WHILE structure, the REPEAT structure, and the LOOP structure.

The syntax of each structure is described below:

1. The decision Structures:
   a. The IF Structure

```
IF <condition> THEN
  <statements>
[ELIF <condition> THEN
  <statements>]...
[ELSE
  <statements>]
ENDIF
```

A one line IF statement is also allowed:

```
IF <condition> THEN <statement>
```

b. The CASE Structure

```
CASE <expression> OF
WHEN <expression list>
  <statements>
[WHEN <expression list>
  <statements>]...
[OTHERWISE
  <statements>]
ENDCASE
```

2. The looping Structures:

a. The FOR Structure

```
FOR <for range> [<step>] DO
  <statements>
NEXT [or ENDFOR] <control variable>
```

A one line FOR statement is also allowed:

```
FOR <for range> [<step>] DO <statement>
```

Note:

```
1)  <for range> means
      <control variable> :=
              <initial value> TO <final value>

2)  <step> means
      STEP <step value>
```

b. The REPEAT Structure

```
REPEAT
  <statements>
UNTIL <condition>
```

c. The WHILE Structure

```
WHILE <condition> DO
  <statements>
```

```
ENDWHILE
```

A one line WHILE statement is also allowed:

```
WHILE <condition> DO <statement>
```

d. The LOOP Structure

```
LOOP
  <statements>
  IF <condition> THEN EXIT
  <statements>
  [IF <condition> THEN EXIT
  <statements>]...
ENDLOOP
```

In the above descriptions:

- <condition> is a numeric expression that is considered FALSE if it evaluates to zero, otherwise it is considered TRUE.
- <statement> is a single line statement.
- <statements> is one or more single line statements and/or multi-line control structures.

COMAL Statements

The CASTLE author may use any of the COMAL single line statements. They are grouped into the following classes: assignment, data, screen I/O, file I/O, and miscellaneous. The syntax of each statement is given below with a page number reference to Lindsay (1983):

1. Assignment Statements (p. 104):

   a. Numeric assignment

<numeric variable> := <numeric expression>

<numeric array name>(<index>) := <numeric expression>

b. String assignment

<string variable> := <string expression>

<string array name>(<index>) := <string expression>

A string specificer on either side of the assignment symbol (:=) may have a substring specification added to it as follows:

(<start position>[:<end position>])

c. Incremental and decremental assignment

<numeric specifier> :+ <numeric expression>

<string specifier> :+ <string expression>

<numeric specifier> :- <numeric expression>

2. Data Statements:

These statements are used to specify and read internally stored data.

a. DATA <list of string and/or numeric constants> (p. 28)

b. READ <list of variable names> (p. 149)

c. EOD - System end-of-data function - It returns TRUE after the last data item has been read (p. 63).

d. RESTORE [<label>] - Resets the data pointer to the first data item in the program, or, optionally, the first data item after the specified label statement (p. 163).

e. Label statement (p. 101)

<label>:

3. Screen I/O Statements:

Normally, output to the screen and input from the keyboard are handled within the UNIT routines. The author may, however, take over these functions from within a lesson control module or in a graphic function by using any of the following statements:

a. CURSOR <row>,<column> (p. 26)

b. PRINT [<at part>][<print part>][<continue mark>] (p. 138)

<at part> is AT <row>,<column>:

<print part> is

USING <image expression>:<numeric expression list>

or <print list>

<print list> can include one or more of the following, separated by a <continue mark>:

TAB(<position>) (p. 192)

<string expression>

<numeric expression>

<continue mark> is

a comma (tab to next print zone)

or a semicolon (tab one space).

c. ZONE <numeric expression> (p. 221)

Sets width of print zone. ZONE 0 is the default value. When used as a function, ZONE returns the current print zone setting.

d. INPUT [<prompt string>:] <variable list> (p. 95)

e. KEY$ returns the value of the last key typed. If the

keyboard buffer is empty it returns CHR$(0) (p. 100).

f. GET$(0,n) waits for n key presses and returns their value as a string of length n. It does not echo the key presses to the screen the way INPUT does. INPUT is terminated by the RETURN key, GET$ is not (p. 82).

4. File I/O Statements:

A CASTLE author may use external files. These may be shared files that are available to all students in the same course, or private files that are unique to each student. File names may have up to ten characters of the same type as unit, lesson, chapter and course names. In the OPEN statement the <file id> for a shared file is SHARED$+<file name> and for a private file is PRIVATE$+<file name>. The following statements are used in file I/O:

a. OPEN FILE <file number>, <file id>, <file type>

<file type> is READ, WRITE, APPEND, or RANDOM <record length> (p. 123).

b. WRITE FILE <file number>[<record>]: <expression list> (pp. 215-218)

<record> is ,<record number>[,<offset>]

c. READ FILE <file number>[<record>]:<variable list> (pp. 151-154)

d. GET$(<file number>,n) returns a string of length n containing the next n bytes from the designated file (p. 82).

e. EOF(<file number>) - System end-of-file function - It returns a value of TRUE after the last item in a sequential file has been read, otherwise it returns FALSE (p. 65).

f. CLOSE FILE <file number> (p. 19)

Note: CASTLE authors must not use the abbreviated CLOSE statement since this will close all files including CASTLE system files.

g. DELETE <file id> (p. 32)

h. PASS <disk operating system command> (p. 133)

i. STATUS returns the value of the disk system error channel (p. 184).

5. Miscellaneous Statements:

a. NULL - does nothing (p. 119)

b. SELECT OUTPUT <type> (p. 171)

<type> is

"LP" - line printer, or

"DS" - data screen.

c. STOP [<message>] will interrupt the CASTLE system execution and return control to the COMAL command mode. It may be used to debug programs since variables may be examined and changed in command mode. Execution may be continued from the STOP statement by issuing a CON (continue) command (pp. 23 and 188).

Note: CASTLE authors should not use the END statement nor leave any STOP statements in a routine

that will be executed by students.

d. // <comment> is used to make comments for internal documentation. A comment may be on a line of its own or may follow any CASTLE statement (p. 156). Blank lines are permitted to enhance readability.

CASTLE Statements

To extend COMAL to fit the role of the CASTLE procedure language, a number of additional statements have been defined. These extension statements are referred to as CASTLE statements. Below is a sample lesson entry procedure to illustrate the uses of all these CASTLE statements and the normal form of this type of procedure. The line numbers would be typical of those used at the beginning of a CASTLE lesson module. They will serve as reference pointers in the discussion that follows.

```
2000 castle
2010
2020 // Course: grammar
2030 // Chapter: review
2040 // Lesson: endings01
2050
2060 // Created: 1983 01 15 10:15
2070 // Revised: 1983 03 24 14:25
2080
2090 // Author: R.W.T. Garraway
2100 // Institution: Division of Educational Research
2110 //   Services, The University of Alberta
2120
2130 proc lesson
2140   get'time(b$(0))
2150   get'date(b$(1))
2160   b$(1):+" "+b$(0)
2170   b$(0):="lesson endings01"
2180   unit'("&welcome")
2190   loop
```

```
2200       case part of
2210       when 1
2220         intro
2230       when 2
2240         verb'endings
2250       when 3
2260         noun'endings
2270       when 4
2280         conclusion
2290       otherwise
2300         exit
2310       endcase
2320       delta'time(time1$,b$(2))
2330       put'scores(sid$,student$,b$(2))
2340       reset'counters
2350       reset'respond'chars
2360       checkpoint
2370     endloop
2380     lesson'completed
2390     lesson'x("endings",2)
2400   endproc lesson
2410
```

The line numbers below 2000 are reserved for the CASTLE system routines. After a lesson module has been loaded, line 2000 is executed which invokes the CASTLE system. In part, this initializes the following system variables:

1. PART - is set to the value of part in the student's restart record. This will be 1 if this is the first entry to the lesson, otherwise it will be the number of the part the student was in at sign off.
2. NEXT'PART - is set to PART + 1. This may be reset to any part number by the author during the execution of a part.
3. SID$ - is set to the student's ID number.
4. SNAME$ - is set to the student's nickname.
5. STUDENT$ - is set to the student's full name.
6. RESPOND'CHARS$ - is set to the complete set of keyboard

characters. This variable defines which keys will be recognized when receiving a student's response. It may be reset by the author to any subset of the available keys.

7. NEXT'LESSON$ - is set to the name of the next available lesson. If the current lesson is the last in the chapter it is set to null. This variable may be reset to another lesson name by the author.
8. NRESP, NOTIME, NRIGHT, NWRONG, NUNREC, and NF(0) to NF(9) - are set to zero. These variables are collectively known as the system counters.
9. TIME0$ - is set to the date/time that the student started the course. The form is YYYY MM DD HH:MM:SS.
10. TIME1$ - is set to the time the student started this session. The form is HH:MM:SS.
11. SPROMPT$ - is set to the standard system input prompt ">". It may be reset by the author to any prompt string of up to 50 characters.
12. SEDIT$ - is set null indicating no default editing of a student's response. The author may assign a list of edit functions to this system variable.
13. STIME - is set to 3600 seconds (1 hour), the system default time limit for a student response. The author may reset this system variable.

After initialization, the procedure 'lesson' is called, line 2130. The comment lines, 2020 to 2110, are the minimal internal documentation.

A line by line explanation of the execution of the procedure 'lesson' follows:

1. 2130 - This is the header statement of a lesson entry procedure.
2. 2140 - GET'TIME is a CASTLE statement that returns the current time in the form HH:MM:SS in the string parameter. Buffer 0, B$(0), is set to the current time.
3. 2150 - GET'DATE is a CASTLE statement that returns the current date in the form YYYY MM DD in the string parameter. Buffer 1 is set to the current date.
4. 2160 - A blank and the current time, buffer 0, are appended to the current date in buffer 1.
5. 2170 - Buffer 0 is reset to the string "lesson endings01".
6. 2180 - This illustrates a call to a unit. The unit name is 'welcome' and the '&' prefix indicates that it is a system library unit. It might, for example, display some message to the student utilizing the contents of buffers 0 and 1 as parameters. The exact nature and required parameters for all system library units would be maintained in system library unit documentation. The same type of documentation would be maintained by authors for their course, chapter, and lesson units. (see also Section F - Instructional Unit Language)
7. 2190 to 2370 - This is a LOOP structure that is repeated until all parts of the lesson are completed or the student requests that the session be terminated (sign

off).

a. 2200 to 2310 - This is a CASE structure used to select the next lesson part to be executed. The value of part is initially set by the CASTLE system immediately after the lesson module is loaded and executed. It is reset by the CASTLE CHECKPOINT statement. This example lesson has four parts: intro, verb'endings, noun'endings, and conclusion. Each of these parts begins with a part entry procedure. For example, the header statement for part 2 would be:

proc verb'endings

This procedure would be executed if part is set to 2 and would be used to control that part of the lesson including the calling of other control procedures and instructional units. Control decisions can be based on the values set in the various system variables. If part attains a value outside the range of part numbers in a lesson, 1 to 4 in this case, then the EXIT statement in line 2300 is executed causing an exit from the LOOP structure to line 2380.

b. 2320 - DELTA'TIME is a CASTLE statement that returns in the second parameter the difference of time between the current time and the time given by the

first parameter. Both parameters are strings of the form HH:MM:SS. If the first parameter is not of this form then the string *ERROR* is returned in the second parameter. The first parameter is unchanged by this statement. In this example the difference between the current time and the time the student started this session is placed in buffer 2.

c. 2330 - This is an example of a call to an author defined procedure. In this example, it would be executed after the completion of each part.

d. 2340 - RESET'COUNTERS is a CASTLE statement that sets to 0 all system counters.

e. 2350 - RESET'RESPOND'CHARS is a CASTLE statement that resets the system variable RESPOND'CHARS$ to the complete set of keyboard characters.

f. 2360 - CHECKPOINT is a CASTLE statement that performs the following important actions:

   1) PART is set to the value in NEXT'PART.

   2) NEXT'PART is incremented by one.

   3) The new value of PART is stored in the student's restart record. If the student elects to sign off during the execution of this part, this value of PART will be recovered at the commencement of the student's next session.

8. 2380 - LESSON'COMPLETED is a CASTLE statement that records in the student's record area that the current lesson has been satisfactorily completed. If all lessons

in a chapter have been thus marked completed then the chapter is marked completed. System lessons could be made available that would display the names of chapters in a course or lessons in a chapter, marking those names of completed sections with an asterisk, say.

9. 2390 - This statement would be executed at the completion of all parts in the lesson. LESSON'X is a CASTLE statement that calls an indexed lesson. In this case lesson 'endings02' would be called. The first parameter is a string literal or expression and the second parameter is a numeric literal or expression in the range 0 to 99. To call a non-indexed lesson the author would use the CASTLE LESSON' statement. It has one parameter, a string literal or expression, which is the lesson name, e.g. lesson'("final exam"). To call the next lesson of the default lesson order use: lesson'(next'lesson$). If the name of a called lesson does not exist then the next chapter is called. A forced call to the next chapter could be produced by the execution of lesson'(""). If there is no next chapter then the course is considered completed and the student is signed off. Both LESSON' and LESSON'X cause the student's restart record to be updated with the names of the new chapter and\or lesson.

10. 2400 - This statement marks the end of the lesson entry procedure.

D. System Variables

System Switch Variables

The system switch variables are set by the UNIT interpreter. When a responce is requested, UNREC is set TRUE and all the other system switch variables are set FALSE. If the response is timed out, OTIME is set TRUE. If any of the categories RIGHT, WRONG, or F(0) to F(9) occurs in a UNIT then the system switch variable of the same name is set TRUE and UNREC is set FALSE.

System Numeric Variables

1. LATENCY - is set to the response latency after each response is accepted.
2. NEDITED - is a vector with indices 0 to 10. It is referenced by the system comparison function CN and the system edit function NUMBR.
3. NEXT'PART - is used to hold the number of the next lesson part to be executed.
4. NF(0) to NF(9) - are system counters. NF(n) is incremented each time the category F(n) occurs in a UNIT.
5. NOTIME - is a system counter. It is incremented each time the category OTIME occurs in a UNIT.
6. NRESP - is a system counter. It is incremented each time a response is requested in a UNIT.

7. NRIGHT - is a system counter. It is incremented each time the category RIGHT occurs in a UNIT.
8. NUNREC - is a system counter. It is incremented if the category UNREC remains TRUE at each response request or when a UNIT is exited.
9. NWRONG - is a system counter. It is incremented each time the category WRONG occurs in a UNIT.
10. PART - is set to the current lesson part number.
11. STIME - is used to hold the default response time.

System String Variables

1. LAST'UNIT$ - contains the name of the last unit called in a lesson.
2. LOWER'CASE$ - contains all the lower case letters. It may be assigned to RESPOND'CHARS$.
3. NEXT'LESSON$ - contains the name of the next available lesson.
4. NUMERALS$ - contains the ten digits, 0 to 9. It may be assigned to RESPOND'CHARS$.
5. PRIVATE$ - contains the file name prefix to be used with files that are private to each student.
6. REDITED$ - contains the student's edited response.
7. RESPOND'CHARS$ - is set to the list of characters that will be accepted in a response request.
8. RESPONSE$ - contains the student's unedited response.
9. SEDIT$ - may be assigned a list of edit funtions, separated by commas, and enclosed in quotation marks.

They are executed in turn after each response is accepted.

10. SHARED$ - contains the file name prefix to be used with files that are shared by all students in a course.
11. SID$ - contains the student's computer id number.
12. SNAME$ - contains the student's nickname.
13. SPROMPT$ - can be assigned a prompt message of up to 50 characters. It is displayed as a prompt each time a response is requested.
14. STUDENT$ - contains the student's full name.
15. SYMBOLS$ - contains all the non-alphanumeric keyboard characters available. It may be assigned to RESPOND'CHARS$.
16. TIME0$ - contains the date and time that the student started the course, in the form YYYY MM DD HH:MM:SS.
17. TIME1$ - contains the time that the student started the current session, in the form HH:MM:SS.
18. UPPER'CASE$ - contains all the upper case letters. It may be assigned to RESPOND'CHARS$.

E. System Functions

COMAL General Functions

The following COMAL functions are available to the CASTLE author. Refer to Lindsay (1983) for detailed descriptions:

1. ABS - absolute value (p. 1)
2. ATN - arc tangent (p. 8)
3. CHR$ - character represented by a given byte value (p. 17)
4. COS - cosine (p. 25)
5. EXP - e raised to a specified power (p. 72)
6. INT - integer value (p. 97)
7. LEN - current length of a specified string (p. 103)
8. LOG - natural logarithm (p. 110)
9. ORD - ordinal byte value of specified character (p. 129)
10. PEEK - byte value at specified memory address (p. 134)
11. POKE - place a specified byte value in a specified memory location (p. 136)
12. RND - random number generator (p. 167)
13. SGN - arithmetic sign of a number as -1, 0, or 1 (p. 177)
14. SIN - sine (p. 178)
15. SPC$ - string of specified number of spaces (p. 180)
16. SQR - square root (p. 182)
17. STR$ - specified numeric converted to string form (p. 190)
18. TAN - tangent (p. 194)
19. TIME - system clock in sixtieths of a second. (p. 197)
20. VAL - numeral in string form converted to a numeric (p. 208)

CASTLE General Functions

Four additional general functions have been defined for the CASTLE procedure language: CNUM, EVAL, PMATCH, and ROUND. These are similar to functions of the same name in NATAL-II. ROUND is called RND in NATAL-II which conflics with COMAL's general function RND, random number (Honeywell, 1981b, pp. 89-92). CNUM and EVAL allow the CASTLE author to extract numeric information from string data, PMATCH compares two strings, and ROUND rounds a number to the nearest integer.

The following are detailed descriptions of each function:

1. CASTLE General Function CNUM:

```
The function cnum may be used by a CASTLE author to convert
a numeric string to a numeral.

Prototype:  return:=cnum(str$,p)

Parameters:
-IN
  str$ - a string containing a numeric of the following
           syntax:
         [+\-][digits][.[digits]][e\E[+\-][digits]]
  p    - a pointer into the string from where to scan to
           find a legal numeric
-OUT
  p    - if a legal numeric is found, points to the position
           just after that numeric
         otherwise points to the position after the end of
           the string

Return value:
  If no numeric is detected, 0 is returned.
  If the numeric would cause an overflow, 1 is returned.
  Otherwise the converted number is returned.

Error messages:
  If the numeric would cause an overflow, the following is
    displayed:
```

castle error - cnum overflow

2. CASTLE General Function EVAL:

The function eval may be used by a CASTLE author to convert a numeric expression in string form to a single number.

Prototype: return:=eval(str$)

Parameters:
-IN
  str$ - a string containing the numeric expression to be evaluated

Return value:
  The string is processed by procedure c'numbr then passed to procedure c'expression for evaluation. The return value from c'expression is returned.

Error messages:
  castle error - division by zero changed to division by one
  castle error - numeral overflow
  castle error - numeral missing
  castle error - missing ')'

Calls procedure c'numbr and function c'expression

Procedure C'NUMBR called by function EVAL:

This procedure changes all non-numeric expression string characters to spaces.

Prototype: c'numbr(str$)

Parameters:
-IN
  str$ - a string of any characters
-OUT
  str$ - a string containing only characters acceptable for evaluation as a numeric expression by function c'expression.

Function C'EXPRESSION is called by function EVAL:

This function is used to evaluate a numeric expression in string form, converting it to a single number.

Prototype: return:=c'expression(str$,p,error)

Parameters:
-IN

```
  str$   - a string containing a numeric expression in
           standard form but ending with a non-numeric,
         - spaces between operators and numerals are
           permitted,
         - the five basic operations (+-*/|) and parentheses
           are allowed
  p      - a pointer to the first character of the expression
-OUT
  p      - points to the non-numeric following the expression
  error  - if division by zero would occur, error is set to 1
           if a numeral would cause an overflow, error is set
              to 2
           if a numeral is missing, error is set to 3
           if a closing parenthesis is missing, error is set
              to 4
           otherwise error is not changed.
```

Return value:
  If no expression is detected, 0 is returned.
  Otherwise the converted expression is returned with the following conditions:
  - if division by zero would occur, division by 1 is executed
  - if a numeral would cause an overflow, it is replaced by 1
  - if a numeral is missing, it is replaced by 0
  - if a closing parenthesis is missing, it is assumed.

Exception:
  If the evaluated expression causes an overflow, a system overflow error occurs and control passes to the COMAL system.

3. CASTLE General Function PMATCH:

The function pmatch compares str1$ to str2$ and returns the percentage of character positions which contain the same characters in both strings.

Prototype: return:=pmatch(str1$,str2$)

Parameters:
-IN
  str1$ and str2$ - the strings to be compared

Return value:
  The percentage of character positions containing the same character.

4. CASTLE General Function ROUND:

The function round returns a number rounded to the nearest integer.

Prototype: return:=round(number)

Parameters:
-IN
  number - the number to be rounded

Return value:
  number rounded to the nearest integer.

System Comparison Functions

These functions are used in UNIT categorization statements to compare a specified target with the student's response. They may also be used in creating user defined comparison functions. The following three functions are currently included in the CASTLE design:

1. CASTLE Comparison Function CC - compare character:

Function cc is the compare character function of NATAL-II. It may be used in much the same way as in NATAL, with the following exception; all parameters must be specified.

Prototype: return:=cc(target$,word,spec$)

Parameters:
-IN
```
  target$ - this is a string literal or expression which is
            to be compared with the edited student response
            in redited$
          - "*" will match any single character
          - "&" will match any string including the null
            string
          - "," is used to separate alternate responses
          - "(" and ")" are used to group alternate
            responses, but nesting these will return
            unpredictable results
          - examples: "a,b" is equivalent to "(a,b)"
                      "&a" will match anything up to the
                         first "a"
                      "&aa,&a" will match "...aa" or "...a"
```

```
                    "&a,&aa" will match "...a" but not
                       "...aa"
                    "a,(b,c)" will return unpredictable
                       results; use "a,b,c" which is
                       equivalent
                    "&(a,b)" will match anything; use
                       "&a,&b"
                    "(&a,&b)c(e,f)&" will match:
                              "...ace..."
                              "...acf..."
                              "...bce..."
                              "...bcf..."
                    "&*abcd" will match anything; use
                       "*&abcd", at least one character
                       will precede "abcd"
                   In the above examples the letters may
                   be replaced by any suitable substrings
                   which could  include "*" or "&" if they
                   are not adjacent to each other or the
                   substring boundary.
  word    - this is a numeric literal or expression which
            indicates at which word in redited$ the
            comparison is to begin
          - if less than 1, the comparison begins at word 1
  spec$   - is a string literal or expression which may be
            used to alter the characters used for "*&(,)" in
            target$
          - if null, no substitutions are made; otherwise up
            to the first five characters of spec$ are used
            as substitutes for "*&(,)" in that order
```

Return value:
TRUE is returned if a match succeeds, otherwise FALSE is returned.

Refers to system variable redited$

2. CASTLE Comparison Function CK - compare keyletter:

Function ck is the compare keyletter function of NATAL-II. It may be used in exactly the same way as in NATAL, except all parameters must be specified.

Prototype: return:=ck(target$,criterion)

```
Parameters:
-IN
  target$ - this is a string literal or expression
            containing one or more words
          - each word represents a word skeleton to be
            matched, in the same word order, with a word in
            the edited student response in redited$
          - example: word skeleton "rcv" would match "rcv",
```

```
            "recieve", "receive", "recover", or
            "xxxrxxxcxxxvxxx"; but not "rvc", "review", or
            "cavort"
  criterion - this is a numeric literal or expression
              indicating the minimum number of words in
              target$ that must be matched
            - if 0, all words in target$ must be matched
```

Return value:
  Return TRUE if the criterion is met, otherwise return FALSE.

Refers to system variable redited$

3. CASTLE Comparison Function CN - compare numeric:

Function cn is the compare numeric function of NATAL-II. It may be used in the same way as in NATAL, except all parameters must be specified.

Prototype: return:=cn(target,tolerance,word)

```
Parameters:
-IN
  target    - is a numeric literal or expression which
              evaluates to the number that is to be matched
  tolerance - is a numeric literal or expression which
              evaluates to the allowable absolute difference
              between the target and the response
            - for matching integer targets and responses
              exactly, it may be set to 0
            - for matching real targets or responses, it
              should be set to a value that allows for
              computer inaccuracies in the ninth least
              significant digit, e.g. for target=4523.34,
              set tolerance=0.00001 to match resposes
              4523.33999 to 4523.34001 inc.
  word      - is a numeric literal or expression that
              indicates which position in the system vector
              nedited contains the response to be compared
            - if word is less than 1, it is set to 1; if it
              is greater than 10, it is set to 10
            - if the system edit function numbr has not been
              called since the last execution of the
              response statement, it is invoked by the cn
              function
```

Return value:
  Return TRUE if nedited(word)=target within the tolerance, otherwise return FALSE.

Calls procedure numbr
Refers to system variables nedited and redited$

System Edit Functions

These functions may be used as general functions in the CASTLE procedure language, e.g. in the creation of a user defined edit function. They are also used, without parameters, with the SEDIT$ system variable and in the UNIT EDIT statement. When used in this way the parameter STR$ refers to the REDITED$ system variable and the parameter VECTOR refers to the NEDITED system vector. The function CHANGE may not appear in a UNIT EDIT statement or be assigned to the SEDIT$ system variable. See the definition of a Character Swap String in the EDIT statement definition in Section F, Instructional Unit Language. The following EDIT functions are defined:

1. CASTLE Edit Function ALPHA:

Procedure alpha changes all string literal numerics (which may contain a decimal point, a sign, or be in "E" notation) to spaces in a given string and then executes the procedure mulsp on the string.

Prototype: alpha(str$)

Parameters:
-IN
  str$ - the string to be operated on
-OUT
  str$ - the changed string

Calls procedure mulsp

2. CASTLE Edit Function CHANGE:

Procedure change replaces one substring with another substring in a given string. This replacement may be for all occurances of the replacement substring or for a given maximum number of occurances in the original string.

Prototype: change(count,a$,b$,str$)

```
Parameters:
-IN
  count - if less than 1, all occurances of a$ in str$ are
             replaced with b$;
          otherwise, a maximum of 'count' occurances of a$
             are replaced
  a$    - substring to be replaced
        - if nul, no replacement is made
  b$    - replacement substring
        - if nul, a$ is deleted from str$
  str$  - string in which specified occurances of a$ are to
             be replaced by b$
        - a$ and b$ may be of different lengths
-OUT
  str$  - the changed str$
```

3. CASTLE Edit Function MULSP:

Procedure mulsp removes all leading and trailing spaces and changes all multiple spaces to single spaces in a given string.

Prototype: mulsp(str$)

```
Parameters:
-IN
  str$ - the string to be operated on
-OUT
  str$ - the changed string
```

4. CASTLE Edit Function NUMBR:

Procedure numbr extracts string numerals embedded in a string, converts them to numerics, and places them in a numeric vector.

Prototype: numbr(dimension,vector,str$)

```
Parameters:
-IN
  dimension - gives the maximum length of vector
  str$      - the string containing the list of numerals
```

- each numeral must be separated from each adjacent numeral by at least one non-numeric

-OUT
  vector
- references a user created numeric vector of dimensions (0:dimension)
- vector(0) returns the number of numerics found in str$
- if str$ contains no numerics, vector(0) will be set to 0
- if a numeric 0 is detected as the 'dimension'th numeric in str$, vector(0) will equal 'dimension'-1
- if str$ is null, vector(0) will be set to 0
- if the number of numerals in str$ is greater than dimension, vector(0) will be set to dimension and only the first 'dimension' numerals will be converted

Calls function cnum

5. CASTLE Edit Function PUNC:

Procedure punc changes the following punctuation characters to spaces in a given string: ' " ( ) , ; : . ? !

Prototype: punc(str$)

Parameters:
-IN
  str$ - the string to be operated on
-OUT
  str$ - the changed string

6. CASTLE Edit Functions SHDN and SHUP:

Procedure shdn may be used by a CASTLE author to shift all upper case characters in a string to lower case. Procedure shup does the reverse.

Prototypes: shdn(str$)
            shup(str$)

Parameters:
-IN
  str$ - the string to be shifted
-OUT
  str$ - the shifted string

7. CASTLE Edit Function SYMB:

Procedure symb changes every character that is not an

alphabetic character to a space in a given string and then executes the procedure mulsp on the string.

Prototype: symb(str$)

Parameters:
-IN
  str$ - the string to be operated on
-OUT
  str$ - the changed string

Calls procedure mulsp

System Graphic Functions

A full set of graphic functions has not been defined for CASTLE. the following graphic functions may be used in creating user defined graphic functions:

1. CASTLE Graphic Function BOX:

   Procedure box draws a box on the display screen.

   Prototype: box(top,bottom,left,right)

   The parameters define the 'top' and 'bottom' rows, and the 'left' and 'right' columns of the box.

2. CASTLE Graphic Function BOX'HLINE:

   Procedure box'hline draws a horizontal line between two vertical lines.

   Prototype: box'hline(row,left,right)

   The line is drawn in row 'row' from column 'left' to column 'right'. It is assumed that vertical lines exist in columns 'left' and 'right'.

3. CASTLE Graphic Function BOX'VLINE:

   Procedure box'vline draws a vertical line between two horizontal lines.

   Prototype: box'vline(column,top,bottom)

The line is drawn in column 'column' from row 'top' down to row 'bottom'. It is assumed that horizontal lines exist in rows 'top' and 'bottom'.

4. CASTLE Graphic Function BOX'WINDOW:

Procedure box'window draws a box on the display screen and creates a display window inside the box.

Prototype: box'window(top,bottom,left,right)

This procedure calls the following two procedures:
box(top,bottom,left,right)
set'window(top+1,bottom-1,left+1,right-1)

5. CASTLE Graphic Function CROSSBAR:

Procedure crossbar draws a crossbar at a character position where a vertical and a horizontal line intersect.

Prototype: crossbar(row,column)

A crossbar is drawn at row 'row' and column 'column' of the display screen. It is assumed that a horizontal and a vertical line pass throught the specified character position.

6. CASTLE Graphic Function HLINE:

Procedure hline draws a horizontal line.

Prototype: hline(row,left,right)

The line is drawn in row 'row' from column 'left' to column 'right'.

7. CASTLE Graphic Function PAGE:

Procedure page clears the currently defined window and places the cursor at the top left corner of the window. If no window is defined the whole screen is considered the currently defined window.

Prototype: page

8. CASTLE Graphic Function SET'WINDOW:

Procedure set'window defines a display window.

Prototype: set'window(top,bottom,left,right)

The parameters define the 'top' and 'bottom' rows, and 'left' and 'right' columns of the window.

9. CASTLE Graphic Function VLINE:

Procedure vline draws a vertical line.

Prototype: vline(column,top,bottom)

The line is drawn in column 'column' from row 'top' down to row 'bottom'.

F. Instructional Unit Language

The CASTLE UNIT is a faithful implementation of the NATAL-II UNIT with a few exceptions that are explained in the statement descriptions affected. For a complete description of the NATAL UNIT see Honeywell (1981b, pp. 27-30 & 61-66). A UNIT may have one DISPLAY statement followed by one RESPONSE statement which in turn may be followed by the UNIT body. A description of each UNIT language statement follows:

1. CASTLE UNIT DISPLAY Statement:

Prototype: DISPLAY ON <device> <text>

The DISPLAY statement passes <text> to the display sub-language interpreter for formatting and display on <device>. Currently two devices are supported, PRINTER and SCREEN. There may be only one DISPLAY statement per UNIT and it must be the first statement. It is optional.

2. CASTLE UNIT RESPONSE Statement:

Prototype: RESPONSE [ON <device>] [<qualifier-list>]

The RESPONSE statement requests a response on <device> under the conditions of the <qualifier-list>. Currently only one device is supported, KEYBOARD. The following qualifiers are supported.

a. APPEND - Each response, when accepted, is appended to the previous response, separated by a space, in the system variable RESPONSE$. The default is not to append.

b. NCHAR=n - The response is limited to n characters. If more than n characters are typed, response acceptance in terminated. The permissible values for n are from 1 to 100. The default value for NCHAR is 100.

c. TIME=n - The response is limited to n seconds. If the response time exceeds n seconds, response acceptance in terminated. The default value for TIME is the value of the system variable STIME.

d. RESPONSE WINDOW (TOP,BOTTOM,LEFT,RIGHT) - This defines the display window, within which the response when typed, will appear. It is always cleared first and SPROMPT$ displayed in the top left corner of the window. The default parameter values are (10,11,1,80).

e. REPRISE WINDOW (TOP,BOTTOM,LEFT,RIGHT) - This defines the initial display window setting for any

REPRISE statement executed in the UNIT body. It is set and cleared immediately after a response is accepted. The default parameter values are (15,23,1,80).

The NATAL qualifiers FONT and TYPE have not been defined in CASTLE. The NATAL qualifier POSN has been replaced by RESPONSE WINDOW. After execution of a RESPONSE statement the following system variables are affected: F(0) to F(9), RIGHT, and WRONG are set FALSE; UNREC is set TRUE; if the response was not in time OTIME is set TRUE and NOTIME is incemented, otherwise OTIME is set FALSE; LATENCY is set to the response time in seconds; NRESP is incremented; and the response is placed in REDITED$ and RESPONSE$. If there are any edit funtions assigned to SEDIT$, they are executed in order before the unit body is executed. There may be only one RESPONSE statement per UNIT and it must follow the DISPLAY statement if there is one. The RESPONSE statement is optional.

3. CASTLE UNIT Body:

It is comprised of any number of COMMENT, EDIT, Categorization, and Reprise statements in any order.

a. CASTLE UNIT EDIT Statement:

Prototype: EDIT <edit-list>

The <edit-list> is made up of two types of edit items: Edit Function references and Character Swap Strings. An Edit Function reference consists of the name of an Edit Function. A Character Swap String consists of two strings: "string1" and "string2". When on EDIT statement is encountered each edit item is executed in turn. For a character Swap String the following occurs. If "string1" is null then "string2" is inserted at the beginning of REDITED$, otherwise the function CHANGE is called with "string1", "string2", and REDITED$ as parameters. All occurances of "string1" in REDITED$ are replaced with "string2".

b. CASTLE UNIT Categorization Statement:

Prototype <category> <target>

<category> may be one of F(0) to F(9), RIGHT, or WRONG. <target> is a boolean expression containing any number of comparison functions as operands separated by one of the boolean operators AND or OR. An operand may be prefixed with NOT and/or replaced by a parenthesized <target> boolean expression. This is a slight difference from NATAL which permits the operand to be replaced by any boolean expression. When a Categorization statement is encountered

<target> is evaluated as TRUE or FALSE and <category> is assigned that value. If evaluated TRUE then the counter for <category> is incremented and UNREC is set FALSE.

c. CASTLE UNIT Reprise Statement:

Prototype: <action> <category> <text>

<action> may be either REINForce or RETRY. <category> may be one of F(0) to F(9), RIGHT, WRONG, UNREC, OTIME, or RENTRY. The NATAL category RTYPE is not supported. RENTRY is set TRUE if the unit was called by a REPEAT statement for the second or subsequent time in a lesson. If <category> is TRUE and this statement is either the last reprise statement for this category in the unit or has not been previously executed in this lesson, then <text> is passed to the display sub-language interpreter for formatting and display on the display screen. On return from the display sub-language interpreter, if <action> is RETRY then control is passed to the UNIT's RESPONSE statement, otherwise <action> is REINForce and control is passed back to the routine that called the unit.

The UNIT Body may only be entered from a RESPONSE statement or through a RENTRY reprise statement.

Invoking the CASTLE UNIT Interpreter:

The following procedures are used by an author to invoke the CASTLE UNIT interpreter. The two "unit" procedures are called when the named unit is to be executed from the beginning of the unit. The two "repeat" procedures are called when the named unit is to be entered at the next RENTRY reprise statement. If the named unit is not one of the most recent ten units called in the current lesson or there is no RENTRY reprise statement in the unit, the unit is executed from the beginning of the unit. The two procedures with the "x" suffix are used to call units with indexed names. Indicies may range from 0 to 99. An index outside this range will cause an error message to be printed and will pass control back to the COMAL system.

```
Prototypes:  unit'(name$)
             unit'x(name$,index)
             repeat'(name$)
             repeat'x(name$,index)
```

Parameters:

-IN

name$ - the name of the unit to be called

- if an indexed call, the base name of the unit to which the index number, converted to two characters, is appended

index - a numeric expression which should evaluate to an integer in the range 0 to 99

- the unit index

G. Display Sub-Language

The display sublanguage is used to format text and graphics on the student's display screen. It may also be used to format text on a printer. Input to the display sub-language interpreter is lines of text to be formatted and command lines containing formatting commands. These lines may be mixed in any order that will produce the desired results.

A command line begins with the control command character. The default control command character is the ampersand, '&'. One or more commands may follow separated by commas. Some commands must be the last command on a command line. This will be pointed out in the following command descriptions:

1. &A - *A*s-is: The following lines of text are displayed as-is. Command lines are thus not considered command lines and are displayed as-is with the exception that &Z is recognized as the terminator of As-is mode. The &A

command must be the last one on a line.

2. &B(top,bottom,left,right) - *B*ox: This command calls the CASTLE Graphic Function SET'WINDOW with the given parameters.
3. &C(row,column) - *C*ursor Positioning: This command executes the COMAL statement CURSOR row, column.
4. &E(top,bottom,left,right) - *E*rase Window: This command calls the CASTLE Graphic Functions SET'WINDOW, with the given parameters, and PAGE.
5. &Fn - *F*ont Selection: Select font number n. The meaning of each font number is implementation dependent. Font 0 is the default font containing upper and lower case alphabetics, numerics and punctuation symbols.
6. &Gn - *G*raphics Display: Call user graphic function n, UGFn
7. &H<text> - *H*ighlight Text: <text> is displayed in reverse video, separated from surrounding text by a blank. This must be the last command on a line.
8. &In - *I*ndent Text: The following text will be indented n spaces from the left edge of the display window. It remains in effect until either an &I0 or &R command is issued.
9. &Kn - *K*eep Lines: If less than n lines remain on the current page, issue an &N command.
10. &Ln - Skip *L*ines: Issue n+1 carriage return / line feeds. &L0 or &L starts a new line.
11. &M<text> - *M*idpoint Text Display: A carriage return /

line feed is issued and <text> is displayed centred between the left and right edges of the current display window. <text> is truncated on the left and right to fit if needed. This must be the last command on a line. &MH<text>, for highlighting text, and &MU<text>, for underlining text, may be used.

12. &N - *N*ew Page: An &W0 command is issued and the PAGE function is called.
13. &P - *P*aragraph: A carriage return / line feed is issued and the following text is prefixed by five spaces.
14. &Qc - The control command character is changed to character c. c may be one of the following: ! " # $ % ' or &.
15. &R - *R*eset: Commands B, F, I, and S are reset to their default settings.
16. &Sn - Line *S*pacing: n carriage return / line feeds are issued after each line of formatted text is displayed. The default setting is &S1. &S0 is not recognized.
17. &Tn - *T*ab: The COMAL function TAB(n) is called.
18. &U<text> - *U*nderline Text: <text> is displayed with each character and included space underlined, separated from surrounding text by a blank. This must be the last command on a line. On some devices underlining may be erased if underlined text is not followed by a blank line on the display.
19. Vn - Display String *V*alue: The contents of buffer n, B$(n), are formatted for display.

20. &V(<image string>,<numeric expression>) - Display Numeric *V*alue: The following COMAL statement is executed:

    PRINT USING "<image string>": <numeric expression>,

    If the value is to be separated from surrounding text by blanks, these must be in the <image string>.

21. &Wn - *W*ait: The display pauses for n seconds. &W0 displays the message 'Press RETURN to continue' on the bottom line of the display screen and waits for the RETURN key to be pressed.
22. &YC<text> - *C*omment: <text> is a comment and is not displayed. This must be the last command on a line.
23. &YD(<display file name>) - *D*isplay External Text: <display file name> is the name of a file containing source text and commands for the display sub-language. It is accessed and interpreted before proceeding to the next line.
24. &YW(row,column,<window file name>) - Preformatted *W*indow: <window file name> is the name of a file containing parameters and text of a preformatted window. 'row' and 'column' indicate the position of the top left corner of the display window. The file contains two parameters: 'number of rows' and 'number of columns'. The SET'WINDOW function is called with the following parameters: ('row', 'row'+'number of rows'-1, 'column',

'column'+'number of columns'-1). The window is cleared and the preformatted text in the file is displayed in the window.

25. &Z - This command terminates As-is mode. It must be the first command on a line.

In the above commands, a page is considered the current display window. The initial display window for a DISPLAY statement is the entire screen less the bottom line which is reserved for system messages. For a Reprise statement the initial display window is the REPRISE WINDOW set in the RESPONSE statement. When a display window has been filled with formatted text, an &N command is automatically issued before more text is displayed.

The numeric parameters in the above commands may be numeric expressions that may reference the system numeric registers, C(0) to C(99), as operands.

The CASTLE Display Sub-Language closely follows the design of the NATAL Display Sub-Language but there are substantial differences (see Honeywell, 1981b, pp. 69-81). These differences have been introduced to simplify the commands and to increase the efficiency of the Display Sub-Language interpreter.

V. Design Specifications for the CASTLE Support System

The CASTLE system is implemented by four support sub-systems: the courseware development sub-system, the courseware presentation sub-system, the registration sub-system, and the performance analysis subsystem.

A. The CASTLE Registration Subsystem

The registration subsystem is needed to identify and keep records on the users of the CASTLE system and the courseware that they will use. There are three parts to this subsystem: course registration, class registration, and system library registration. System programs are used to maintain these records. These programs allow for the addition, deletion, reviewing and updating of records. For the protection of these system records, access to the various levels of the registration subsystem is restricted by passwords.

Course Registration

Each course to be developed or accessed by students must be registered. For each registered course the following must be maintained:

1. An entry of the course name and its internal reference code in the system's course names file.
2. A course header file. In this file is maintained:
   a. A password. This must be used by the author to gain access to course record maintainence routines and to

the courseware development system to develop lesson modules and instructional units for the course.

b. The access status. A course may be locked, indicating that it is under development and may not be used by a class of students. An unlocked course may also be permitted for browsing. That is, a non-registered student can have access to the course using the courseware presentation browse mode.

c. The default chapter order. This is a list of internal chapter codes indicating the author's preferred chapter order.

d. The course internal documentation. This should include but not be limited by:

1) The course creation date.

2) The course's long descriptive title.

3) The course author(s).

4) The institution of origin.

5) The existance of external documentation.

e. A course chapter names record.

For each chapter of a course the following must be maintained:

1. An entry of the chapter name and its internal reference code in the course chapter names record of the course header file.

2. A chapter header file. In this file is maintained:

a. The default lesson order. This is a list of internal chapter codes indicating the author's preferred

lesson order for the chapter.

b. A chapter lesson names record. The name of each lesson in the chapter and its internal reference code is maintained in this record.

Class Registration

An instructor who wishes to use a CASTLE course with a group of students must register the class and the students with the CASTLE system. The class name is the course name with a two digit class number appended. For each class the following must be maintained:

1. An entry of the class name and its internal reference code in the system's class names file.
2. A class header file. In this file is maintained:
   a. A password. This must be used by the instructor to gain access to class record maintenance routines and performance analysis routines for the class.
   b. A list of the computer ID numbers of all students in the class.
   c. The class chapter order. This is a list of internal chapter codes indicating the default chapter order for students in the class. This could be a direct copy of the course author's preferred chapter order from the course header file. The instructor may, however, rearrange this chapter order, or insert chapters from other courses or from the system library. If chapters from other courses are to be

used, these courses must be registered with the system and a list of pointers to these external chapters maintained in the class header file.

d. The class internal documentation. This should include but not be limited by:

1) The class creation date.

2) The class instructor.

3) The institution to which the class belongs.

3. A student registration file for each student. The following records are maintained in each student registration file:

a. A password. This must be used by the student during the sign-on procedure at the beginning of each computer session.

b. A review mode permit flag. This flag is set on if the student may use the review courseware presentation mode.

c. A current mode flag. This flag is set on if the student is currently in the review mode and set off if under direct courseware control (controlled mode). At registration this flag is set off.

d. The chapter order. This is a list of internal chapter codes indicating the chapter order for this student. It could be a direct copy of the class chapter order or the order may be modified by the instructor and could include external chapters linked to this class

e. A lesson completion map. This is a boolean two dimensional matrix. Each row represents a chapter in the order of the student's chapter order record. Each column represents a lesson in the order of the chapter author's default lesson order in the chapter header file.

f. A controlled mode restart record. The contents of this and the following record are described under the courseware presentation subsystem.

g. A review mode restart record.

System Library Registration

The system library may contain chapters, lessons, units, displays and windows as identifiable elements. A system name file is maintained for each type. These system elements are developed and tested by course authors. When ready for entry to the system library, the internal documentation for the element is updated to indicate that it is now in the system library, and the system library registration routine is called to rename the element and to place its name and internal reference code in the appropriate system name file.

B. The CASTLE Courseware Development Subsystem

An author creates a course by developing lesson modules in the CASTLE procedure language and instructional units in the unit language and display sub-language.

Lesson Module Development

When a lesson name is registered, the sytem creates a lesson module which contains the CASTLE run-time routines, internal documentation statements, and the skeleton of the lesson entry procedure. A lesson module is stored on disk as a COMAL program file. The courseware development subsystem, at the request of an author, will load any desired lesson module into the active workspace for editing. All COMAL system commands are available to the author while editing a lesson module.

Below is the skeleton lesson module provided by the registration subsystem.

```
2000 castle
2010
2020 // Course:
2030 // Chapter:
2040 // Lesson:
2050
2060 // Created:
2070 // Revised:
2080
2090 // Author:
2100 // Institution:
2110
2200 proc lesson
2300   loop
2400     case part of
2500     when 1
3000     otherwise
3100       exit
3200     endcase
3300     checkpoint
3400   endloop
3500   lesson'("")
3600 endproc lesson
```

The only program statement in a CASTLE lesson module is line 2000 which calls the 'castle' initialization procedure which in turn calls the 'lesson' entry procedure. To test a lesson module the author simply issues the COMAL RUN command. In testing a module, care must be taken not to execute a lesson call statement as in line 3500, since this would load in another lesson module and the current one would be erased. Placing an exclamation mark (!) just after the line number will make the statement a comment which will not execute. The comment symbols may be erased before the module is saved on disk.

The COMAL language is implemented as a three-pass semi-compiler. The first pass is in the COMAL editor which checks each statement as it is entered for correct syntax. The second pass occurs when the RUN command is issued. A complete scan of the program is made to ensure that all procedures, functions, and control structures are properly closed. The third pass executes any program lines. If there are no program lines, just procedures and functions, then the third pass does nothing. Since line 2000 is the only program line in a CASTLE lesson module, it can be made a comment so that the RUN command will simply execute the second pass checks.

To save a lesson module to disk an author performs the following steps:

1. Make line 2000 a comment.
2. Issue the RUN command.

3. Type save'lesson and press RETURN.

   save'lesson is a CASTLE routine that:

   a. Restores line 2000 as a non-comment line.

   b. Updates line 2070 with the current date and time.

   c. Replaces the old copy of the lesson module on disk with the current updated copy.

   d. Loads and runs the courseware development command routine.

The COMAL SAVE command should not be used.

The following COMAL commands may be used by the CASTLE author in editing a lesson module (Lindsay, 1983):

1. AUTO [<starting line number>] [,<line increment>] (p. 9)

   This command creates program line numbers automatically beginning with <starting line number>. Each successive line number is incremented by <line increment>. The default <starting line number> is the highest line number currently used plus 10. The default <line increment> is 10. Null lines may be entered to increase the readability of listings. To exit the AUTO line numbering mode press the STOP key. Line numbers may also be entered manually.

2. DEL <line number range> (p. 30)

   This command deletes from the workspace those lines included in <line number range>. <line number range> may be:

   a. a single line number,

   b. <line number> -, which is the range from <line

number> to the last line number,

c. - <line number>, which is the range line 1 to <line number>,

d. <beginning line number> - <ending line number>.

The DEL command is the only way to delete lines.

3. EDIT [<line number range>] or

EDIT <procedure or function name> (p. 43)

This command lists the entire workspace, the <line number range>, or the named procedure or function to the screen without 'pretty print' indentation of structures.

4. LIST [<line number range>] or

LIST <procedure or function name> (p. 107)

This command is the same as the EDIT command except that lines are indented to show the program structure. To list a CASTLE lesson module, but not the CASTLE run-time routines, to the printer use:

a. select [output] "lp"

b. list 2000 -

When the listing is completed the system will automatically issue a select output "ds".

5. RENUM [[<old beginning line number>;] <new beginning line number>] [,<line increment>] (p. 160)

This command will renumber the lines from the <old beginning line number> to the end of the program starting with the <new beginning line number> and incremented by <line increment>. The default for <old beginning line number> is 1, for <new beginning line

number> is 10, and for <line increment> is 10. Since the 'castle' statement should always be on line 2000 and the revised date/time on line 2070, the standard RENUM command for CASTLE authors would be RENUM 2000;2000. For renumbering lines greater that 2070 an example command might be RENUM 2210;3000,100. All lines from 2210 to the end of the workspace would be renumbered 3000, 3100, 3200, etc.

6. SIZE (p. 179)

   This command displays the size in bytes each of the program and the data, as well as the number of bytes still available in the workspace.

The following COMAL commands should not be used by the CASTLE author because their functions have been replaced by CASTLE routines: BASIC, CAT, CHAIN, DELETE, LOAD, NEW, PASS, SAVE, VERIFY.

Instructional Unit Development

The instructional unit development sytem has three major parts: the UNIT file manager, the UNIT editor, and the UNIT tester. The CASTLE author enters the unit development system from the courseware development command routine by selecting the UNIT file manager.

Each instructional unit is stored as two disk files. One file is a table of integers called the parse table. Each entry represents either an internal reference value, an operation code (opcode), an immediate integer value, a

pointer to another part of the parse table, or a pointer to a record in the other file. The second file, called the string table, is a random access file of string records with a maximum length of 78 characters. Each record holds text data or commands for the display sub-language interpreter, or numeric expressions in string form that are evaluated at run-time. The unit is executed by the CASTLE UNIT interpreter. It carries out the actions specified by the opcodes in the parse table.

1. The UNIT File Manager:

   On entering the UNIT File Manager the author selects a unit directory for the course global units, a particular chapter's units, or a particular lesson's units. A list of the current units of the selected directory is displayed. The following menu items may then be selected:

   a. copy a unit - A unit from an external source may be copied to the current unit directory.
   b. delete a unit - A unit may be deleted from the current unit directory and its disk files erased.
   c. edit a unit - The UNIT Editor is called for a selected unit.
   d. get a new unit directory - The author may select a different unit directory.
   e. history of a unit - The date/time of unit creation, the date/time of the last unit update, and the unit header comment are displayed for a selected unit.

f. list unit names - The list of unit names in the current directory is displayed.

g. quit - Control is returned to the courseware development command routine.

h. rename a unit - The name of a unit may be changed.

i. start a new unit - The author registers the name of a new unit for this directory, its disk files are initialized, and a permanent header comment by the author is placed in the unit header record.

j. test a unit - The UNIT Tester is called for a selected unit.

2. The UNIT Editor:

The UNIT Editor utilizes the principles of incremental compilation. That is, a unit only exists as a set of internal codes in the parse table. At run-time these codes are decoded by the UNIT Interpreter which carries out the specified actions as an instructional interaction with a student. The UNIT Editor is used by the CASTLE author to create, examine, and modify these internal codes. However, the author is never aware of the actual codes but only of their meaning as given in the UNIT Language specification.

The use of incremental compilation has two advantages. At run-time an interpreter can execute the parse codes in the parse table much faster than if it had to parse out the meaning from UNIT language source text. Secondly, the special editor helps the author

create and modify a unit by presenting option as selections from a menu and interrogating for specific parameters. At each step of the process the editor builds error-free code in the parse table.

The UNIT Editor command level presents the author with some current statistics on the unit and a menu of options. The statistics are the amount of free space in the parse table and the number of records still available in the string table. The menu options are as follows:

a. display statement editor - This selects a sub-editor for the DISPLAY statement. All sub-editors either interrogate the author for statement parameters or present options via menu selection. The <text> portion of the DISPLAY statement is edited by the CASTLE System Text Editor. This editor displays a window of up to 20 lines of the <text> and offers the following menu selected options:
   1) back - Move the display window back 15 lines.
   2) copy - Copy a range of lines and insert them before a specified line.
   3) delete - Delete a range of lines.
   4) edit - Enter the screen edit mode. This mode allows the free movement of the cursor in the current display window and the changing, deletion, and insertion of characters on the display. The changes to a particular line are

recorded when the RETURN key is pressed while the cursor is anywhere on that line. To exit the screen edit mode the STOP key is pressed before pressing the RETURN key. The window is redisplayed to confirm the changes.

5) first - Display the first 20 lines of <text>.
6) insert - Insert lines of text before a specified line. Any number of lines may be inserted. To exit the insert mode enter a line containing only a period (.) in the first character position.
7) last - Display the last 20 lines of <text>
8) move - Move a range of lines and insert them before a specified line.
9) next - Move the display window forward 15 lines.
10) quit - Exit the text editor.
11) search - A search is initialized for a specified string within the <text>. If the search is successful a window is displayed having as its first line the line containing the found string. The editor then enters the screen edit mode placing the cursor on the found string. On leaving screen edit mode the author may select to continue searching for the same string or exit to the text editor menu.

It should be noted that all of <text> resides in records in the unit string table on disk. These

records are read, created, and updated directly on the disk by the text editor. Because of limited main memory on most microcomputers, keeping the string table on disk allows it to grow much larger than it could otherwise. This is especially important in CAI when text is used extensively.

b. exit to unit file manager - Return control to the UNIT File Manager.

c. list unit to screen - The parse table and string table are scanned by the editor to generate and display the unit's statements in human readable form.

d. print unit on printer - This performs the same function as the previous item except that the listing is sent to the printer.

e. quit - Control is returned to the courseware development command routine.

f. response statement editor - This selects a sub-editor for the RESPONSE statement.

g. test this unit - The UNIT Tester is called for this unit.

h. unit body editor - This selects a sub-editor for the statements in the unit body. The author is able to step forward and backward through the statements in the unit body. Only the statement type (EDIT, Categorization, Reprise, COMMENT, END OF UNIT) is displayed. At any step the author may select any of

the following options:

1) back - Move back one statement.

2) delete - Delete the current statement.

3) end - Go to the END OF UNIT statement.

4) first - Go to the first statement.

5) insert - Insert a statement before the current statement.

6) list - List the current statement in full.

7) modify - This allows the author to make modifications to the current statement.

8) next - Move forward one statement.

9) quit - Exit the unit body editor.

The <text> portion of any Reprise or COMMENT statements is created and modified by a call to the CASTLE System Text Editor.

3. The UNIT Tester:

This is a special CASTLE lesson module that allows an author to execute a single unit. Before and after each test run of a unit the author may examine or change the contents of system variables, counters, and registers. After each test run the following options are presented to the author:

a. call unit file manager - Return control to the UNIT File Manager.

b. edit this unit - The UNIT Editor is called for this unit.

c. quit - Control is returned to the courseware

development command routine.

d. rerun this unit - Execute the unit again.

Display and Window File Development

A separate file manager is available for both Display and Window files. The options in these routines are principally the same as those in the UNIT File Manager. The editor called from the Display File Manager is the CASTLE System Text Editor. The Window File Editor allows the author to create a preformatted window, mark its boundaries, and record it, along with its parameters, to a window file.

C. The CASTLE Courseware Presentation Subsystem

Courseware is presented under two modes: controlled and review. Under controlled mode the student follows a path through the course controlled completely by the decision algorithms defined by the author in the courseware. If review mode is permitted, the student may interrupt the normal actions of the system and request a transfer to a different place in the course. When the student leaves review mode a return is made to the beginning of the part of the lesson last in under controlled mode.

The users of a course are either registered or non-registered students. Registered students are members of a class who utilize a special version of a course for that class. Restart records are kept for these students and performance records may also be made. If a course is

permitted for browsing then a non-registered student may sign on to that course. Review mode is automatically permitted to non-registered students, but since no restart records are kept, the student is returned to the beginning of a course if an exit is made from review mode. Performance records are not made for non-registered students.

Restart records for registered students are kept under both controlled and review mode. If a student signs off a session while in review mode, this is noted in the student's record area. At the next sign on the review restart record is used to restart the course and the student remains in review mode.

A restart record contains the current values of the following items:

1. the internal chapter code
2. the internal lesson code
3. a table of the names of the last ten units called from the lesson
4. an index to this table indicating the last unit loaded
5. a matrix of values associated with the execution environment of these ten units.
6. the system local registers
7. the sytem counters
8. the system variables NEXT'PART, PART, STIME, LAST'UNIT$, RESPOND'CHARS$, SEDIT$, and SPROMPT$

The sign on routine interrogates the student for the name of a course (for browsing) or the name of a class. If a

course name is given and that course exists and is permitted for browsing, the non-registered student is sent to the beginning of the course.

If a class name is given the student is interrogated for a sign on ID. If this ID exists then the password is requested. Up to three tries are permitted to type a correct ID and password. If not successful the student must restart the sign on procedure. After a successful sign on the student is placed at the beginning of the lesson part that was being executed at the last sign off, or at the beginning of the course if this is the first sign on.

The student may interrupt course execution by pressing the STOP key. The interrupt request is serviced at the next response request or the next time the words "Press RETURN to continue" are displayed at the bottom of the screen. A list of prompt words, with their first letters in reverse video, is displayed at the bottom of the screen. The student may select an action by pressing the key for one of the highlighted letters. To cancel the interrupt the student presses the RETURN key. The following is an explanation of the use of the possible prompts:

1. ?help - An explanation of how to use selected prompts is given.
2. calculator - This prompt is permitted at the author's discretion. The student may use the computer as a calculator by typing in algebraic expressions for evaluation. If the STOP key was pressed during a

response request, the student may have a calculated result returned as an answer to the response request.

3. exit'review - This prompt is displayed only if the student is in review mode. If selected the student leaves review mode and is sent to the beginning of the lesson part that was being executed when review mode was entered.
4. glossary - If permitted by the author, a course glossary look up routine is entered.
5. hint - This may appear only when the interrupt is serviced at a response request. The author may create a 'hint' unit, to be associated with any regular unit, to provide additional instruction to the student.
6. message - The student may send a message to the instructor.
7. postbox - This prompt is displayed only if a message exists for the student from the instructor. Once read, the message is erased from the student's record area.
8. quit - The student wishes to sign off.
9. review - This prompt is displayed only if review mode is permitted. The student may request a transfer to a different part of the current lesson, a different lesson of the current chapter, or a different chapter.

D. The CASTLE Performance Analysis Subsystem

An insturctor may request that performance records be kept for the students. For each student a performance file is created. At the execution of a UNIT Reprise statement or an END OF UNIT statement a performance record is appended to the student's performance file. The following items are included in a performance record:

1. the system switch variables
2. the system counters
3. the following system variables: LATENCY, PART, LAST'UNIT$, and RESPONSE$
4. the internal chapter code
5. the internal lesson code

No routines to access or analyze these performance records have been defined. Some possible routines may be seen in Honeywell (1981c).

VI. Implementation of the CASTLE System

Factors that are generally considered critical in the implementation of a complex system on a microcomputer are internal (RAM) and external (disk) memory limitations and the execution time of system activities. To investigate the suitability of the CASTLE design for use on a microcomputer system, a test implementation of a subset of the CASTLE language and Support System was carried out. The subset represents those aspects of the CASTLE Language and Courseware Development and Presentation Subsystems necessary to demonstrate the creation and delivery of individual lessons. Using this implementation, measurements of memory utilization and execution speeds were made. From these measurements, memory and speed parameters for a full implementation of the CASTLE System were estimated.

A. Hardware Selection

The decision to use COMAL-80 as the system development language for the CASTLE system, necessitated the selection of hardware that supports this language. In North America, COMAL is only available on microcomputers produced by Commodore Business Machines. CBM COMAL-80 Revision 0.12, operates on the PET 4032 and the CBM 8032 microcomputers. It only leaves 5590 bytes of memory free for programs and data. However, the CBM 8096 microcomputer runs the full extended version, CBM COMAL-80 Revision 01.02, providing a workspace of 38692 bytes. Thus, this version of COMAL and the CBM 8096

formed the basis of the development system.

The CBM 8096 is controlled by a 6502A microprocessor and contains 98304 (96K) bytes of read/write (RAM) memory and 18432 (18K) bytes of read only (ROM) memory. 14K of this ROM is occupied by Commodore BASIC 4.0. The remaining 4K is the operating system which supports I/O to a CRT screen, a keyboard, two tape drives, an IEEE-488 parallel communications bus, and a special user port. The operating system also has a built in machine language monitor.

The CRT (display) screen and the keyboard are part of the main unit. Twenty-five lines of eighty characters in two predefined fonts can be displayed on the green phosphor CRT. Each character is defined in a rectangle of 8 by 8 pixels giving a screen resolution of 640 by 200 pixels. The screen lines may be displayed in an open format, the default, which provides two blank pixel rows between each line, or closed format, with no inter-line spacing. The default font containts upper and lower case letters, numerics, 28 symbols, and 38 graphics characters. The alternate font replaces the lower case letters with 26 further graphics characters. A full business type keyboard with a separate numeric keypad is provided. It has special function keys that provide for cursor movement, screen editing, program interruption, and the reverse video display of selected characters.

The IEEE-488 bus is used to connect intelligent peripherals to the main unit. An intelligent peripheral has

its own CPU, operating system, and memory, and can function independently of the 8096. Two such peripherals are needed to round out the CASTLE hardware needs: a disk unit and a printer. A Commodore 4040 disk unit was used. It has two disk drives with a storage capacity of 174848 bytes (organized into 683 blocks of 256 bytes) per drive. There may be 144 directory entries per diskette. The 4040 operating system (DOS 2.0) supports direct (random) access files which are required for the UNIT string files. Access to records in these random files is very fast. A maximum of two disk reads are required to retrieve any record.

A commodore 4022 printer was used as the printing device. It can print up to 80 characters per line with variable lines per page at different line spacing. All characters that can be displayed on the CRT can be printed including reverse video (white characters on a black background). The printer can be set in page mode so that after 60 lines are printed an automatic form feed is performed before the next line is printed. With the printer, hard copy listings of lesson modules and instructional units are easily obtained as are reports generated from student sessions.

B. The CASTLE Test Implementation

Since the Registration Subsystem is not part of the test implementation, the COMAL disk catalogue system is used to register lesson modules. All lesson and unit names are

global and the use of the identifier prefixes '&', '#', '%' is not supported. Display and window files and global registers are not implemented.

All CASTLE statements, variables and functions are implemented with the exception of the following:

1. statements - CHECKPOINT, DELTA'TIME, LESSON', LESSON'X, LESSON'COMPLETED

   Note: To call another lesson from within a CASTLE lesson module, CHAIN "<lesson name>" should be used.
2. variables - NEXT'LESSON$, PRIVATE$, SHARED$, SID$, STUDENT$, TIMEO$
3. functions - the graphic functions, including UGFn

The unit interpreter is completely implemented. Currently only AS-IS mode is supported by the display sub-language interpreter.

The following system programs were developed as part of the CASTLE test implementation:

1. BOOT COMAL - This is the first file on the CASTLE system disk. The system disk always resides in drive 0 of the disk unit while a data disk containing all lesson and unit files is in drive 1. By pressing and releasing the RUN key while holding down a SHIFT key, the BOOT COMAL program is loaded and executed. It sets the printer to upper and lower case printing and turns on page mode. It then loads the COMAL system from the disk in drive 0 and initializes it.
2. CASTLE - This is the CASTLE system initialization

program. It is loaded and executed by the operator issuing the command CHAIN "CASTLE". The operator is queried for the current time and date, then presented with the following options:

a. exit to COMAL - to develop or execute a lesson. A student may execute a lesson by typing the command CHAIN "<lesson name>".

b. call CASTLE unit development system - The UNIT File Manager is loaded and executed.

3. LESSON - To start a new lesson, an author LOADs this file. This is a lesson module which contains the CASTLE run-time routines, internal documentation statements, and the skeleton of the lesson entry procedure. The CASTLE command SAVE'LESSON is not implemented. An author must use the COMAL command SAVE "@1:<lesson name>" to save or resave a lesson module. To retrive a previously SAVEd lesson module for further development the author would use LOAD "<lesson name>".

4. CAL - This is the UNIT File Manager. It is called by CASTLE, or an author may invoke it by issuing the COMAL command CHAIN "CAL". The only UNIT directory supported is for global units.

5. UNIT EDITOR - This is the incremental compilation editor for the UNIT Language. It also contains the CASTLE System Text Editor.

6. UNIT TESTER - This is a special CASTLE lesson for testing a single specified unit. The unit name is

selected in and the UNIT TESTER called from either the UNIT File Manager (CAL) or the UNIT EDITOR.

The last three files are implemented as described in Chapter V, Section B. However, the QUIT option in all three programs causes an exit to the COMAL command level.

C. The UNIT Tables and Codes

As was described in Chapter V, Section B the CASTLE UNIT exists only as internal integer codes in the unit parse table with text, display sub-language commands, and numeric expressions kept in the unit string table.

The string table always resides on disk as a random access file. It is modified directly on the disk by the UNIT Editor and the CASTLE System Text Editor. Its records are directly accessed by the UNIT Interpreter and the Display Sub-language Interpreter. In this implementation the maximum number of string table records is 3000. During editing a forward pointer, a backward pointer and a parse table reference pointer are kept in memory for each string table record. The size of this pointer table is held to 3000 records because of internal memory limitations. When records are added, inserted or deleted, these pointers are updated. At the conclusion of an editing session, the string table is re-assembled on disk from the information in the internal pointer table. The first record always contains the unit's creation date/time, its last revision date/time, and a permanent 30 character header comment.

The parse table is stored on disk and kept in RAM memory as an integer vector with a maximum length of 1000. The length of 1000 was arbitrarily chosen in an attempt to provide adequate storage for the code of even a fairly large unit while still conserving as much memory as possible. Both the editing and execution of the parse table is carried out in main memory. COMAL provides for the transfer of entire arrays, which includes vectors, between internal RAM and disk which is very fast and straight forward.

The parse table is divided into a fixed part of 34 integers and a variable part of one or more integers. The fixed part contains counters of the number of integers in the parse table and the number of records in the string table, offsets to the code in the parse table of the first Reprise RENTRY statement and the last Reprise statement in each category, and the parameters for the DISPLAY and RESPONSE statements. The variable part contains the code for all the statements in the unit body.

D. The Implementation Parameters

Listed below are the CASTLE system programs developed for this implementation. After each, the following is given: the disk storage space the program uses in blocks, the amount of memory the program requires (in bytes) for its own storage and for its data structures, the amount of memory storage remaining, and the time in seconds taken to load the program from disk.

1. BOOT COMAL - 1 block, the remaining parameters are not applicable
2. COMAL-INTERPRETER - 217 blocks, 54528 program bytes, 5084 data bytes, 38692 bytes free, 33.0 seconds.
3. CASTLE - 5 blocks, 1008 program bytes, 148 data bytes 37536 bytes free, 3.2 seconds.
4. LESSON - 66 blocks, 14156 program bytes, 9176 data bytes, 15360 bytes free, 14.0 seconds.
5. CAL - 23 blocks, 4947 program bytes, 5004 data bytes, 28741 bytes free, 10.3 seconds.
6. UNIT EDITOR - 77 blocks, 17381 program bytes, 17113 data bytes, 4198 bytes free, 21.2 seconds.
7. UNIT TESTER - 70 blocks, 15210 program bytes, 9278 data bytes, 14204 bytes free, 15.9 seconds.

The time to load LESSON is the minimum load time for any lesson module. Since an author is free to add many other procedures and functions to the basic LESSON, lesson modules will vary in length and additional load time. The time to load each additional 1024 bytes is 1.0 seconds.

During student execution of a CASTLE lesson the following timings are also critical:

1. the initialization of the lesson - 1.1 seconds on average
2. the initialization of a unit from the moment it is called until the first DISPLAY <text> appears on the screen - 2.5 seconds on average
3. the answer analysis response time from the moment the

student enters an answer by pressing the RETURN key until the Reprise <text> appears on the screen - from 2.1 seconds for simple response analyses to 7.7 seconds for the more complex response analyses.

In a full implementation of the CASTLE system there would, of course, be additional system programs for the Registration and Performance Analysis Subsystems. The CAL (UNIT File Manager) program would be expanded so that it could reference other UNIT directories. There should be no memory limitations for these system extensions.

Further run-time routines would be added to the LESSON program for accessing student, class and course records, and to complete the Display Sub-language Interpreter. Approximately 2000 bytes of additional program code would be needed. This would reduce the memory available to authors for lesson module procedures and functions. Accessing registration records at lesson initialization would probably add 3.0 seconds to this operation. Other timings should not be critically affected.

VII. Conclusion

A. Evaluation

In Chapter III, Section A some requirements for a CAI system and courseware development were specified. One criterion was to provide for the separation of instructional strategy from subject content. The CASTLE system makes this provision.

The general instructional strategy is controlled by the CASTLE procedure language in the lesson modules. Specific control of an instructional interaction is through the order of statements in the UNIT Body. Subject content is presented via the UNIT DISPLAY and Reprise statements and Display and Window Files. Other content data, such as target parameters for comparison functions and variables for the Display Sub-language could be assembled in shared data files and read into buffers at run-time.

A second criterion was to provide for the needs of the courseware designer. Listed below are the needs that were mentioned in Chapter III and how they are met by the CASTLE system:

1. easy control of branching and return, e.g. going to a glossary or a help sequence at the student's request. If permitted by the author the student can interrupt the lesson by pressing the STOP key and select to view a course glossary or a special 'hint' sequence. The screen

content is automatically saved before branching and restored on return and the student continues from the point at which the STOP key interrupt was serviced.

2. score keeping and internal monitoring. The CASTLE language provides system counters and switches for the CASTLE categories F(0) to F(9), RIGHT, WRONG, OTIME and UNREC, and variables which return the number of responses made, the response LATENCY, the unedited and edited response and the sign-on time. The author has access to the current lesson part number and may dynamically set the next part number and the next lesson name.
3. monitoring of student progress. The following information in each student's record area would assist with this monitoring:
   a. the current presentation mode, controlled or review
   b. the lesson completion map
   c. the current chapter for each mode
   d. the current lesson for each mode
   e. the current part for each mode
   f. the date/time the course was started
   g. other accounting information, i.e. number of sessions, sign-on date/time of last session, total time of last session, total accumulated session time, could be added.

A fully developed Performance Analysis Subsystem would add a great deal to the monitoring of student progress.

4. access to numerical calculating power. By using the STOP key interrupt and selecting calculator mode, the student may perform arithmetic calculations and return an answer to the system. An extended calculator could give access to system functions and variables.
5. flexible input. The author may define which keyboard characters will be recognized during a student's response. A default response time limit may be set as well as a specific time limit for each response. The number of response characters may also be limited. While typing in a response the student may insert or delete characters from the response before submitting it for analysis.
6. good answer analysis. Three of the six NATAL comparison functions are provided: compare character (CC), compare keyletter (CK), and compare numeric (CN). These, plus up to 10 user defined comparison functions, may be combined in a boolean expression of any complexity as part of the UNIT Categorization statement.
7. easy creation of graphics and animation. This criterion is not met in the current CASTLE design.
8. easy procedures to review and test courseware. Listings of individual lesson modules and instructional units are easily obtained from the system and may be used to review a course. Listings of the default chapter order or the default lesson order may be retrieved from the Registration Subsystem. Lesson modules are created in an

interactive environment and may be tested without entering a course as a student. A special UNIT Tester is available for testing individual units.

9. tailoring courseware to individual student needs, e.g. use of external parameters to adjust pathway through a course. Each instructor may specify the chapter order for each class and may even specify it separately for each student. Within a lesson the author may dynamically set the number of the next part as well as dynamically select the next lesson.

The last criterion is that the system should be "user friendly" and expandable. Lesson modules are developed under the COMAL command interactive environment. This provides for automatic line number generation, full screen editing, syntax checking on input, verbose meaningful error messages, the listing of procedures and functions by name, multi-line deletion, and immediate execution. Units are developed by incremental compilation techniques with options presented via menu selection and interrogation for individual parameters. The system text editor provides a simple set of powerful commands via menu selection and full screen editing of text. The technique of presenting options via menu selection and interrogating for parameters is carried through all system support routines.

A rich variety of intrinsic functions is provided: twenty COMAL and four CASTLE general functions, three comparison functions, eight edit functions, and nine graphic

functions. The experienced designer or researcher may extend the language with user defined general, comparison, edit, and graphic functions.

The use of COMAL as a system development language proved most successful. System programs, procedures and functions were easily developed and debugged using standard top-down design with step-wise refinement. Though the system lacks a trace feature, this did not retard development work since each procedure and function developed was relatively small and could be tested as soon as it was coded. The COMAL feature of storing sub-program modules on disk to be later appended to other programs in memory independent of original line numbering proved most useful.

The COMAL editor lacked two features that would have been helpful: the ability to list just the header statements of all procedures and functions in the workspace and the ability to scan the workspace for a specified string with the option of changing it to another specified string.

The development of the CASTLE system in the COMAL environment seemed much easier than the author's experience of writing a compiler in Pascal.

B. Recommendations

The test implementation has demonstrated the viability of the CASTLE design for use on microcomputer systems. Two possible research and development paths, to run in parallel, might be defined. The first would examine the use of the

system by both experienced and inexperienced authors. The results would be used to define modifications for the enhancement of the system. This research path would also examine the use of the system by instructors and students.

The second path might define further extensions and developments for the system. The following outlines a proposed program:

1. The test implementation would be extended to include all of the current design.
2. The design of the CASTLE Performance Analysis Subsystem would be completed and implemented.
3. The remaining NATAL comparison functions, compare algebraic (CA), compare phonetic (CP), and compare graphic (CG) would be implemented.
4. The system could be extended to a multi-user environment, through a local network and shared I/O devices.

The new Commodore 64 is an inexpensive and popular microcomputer that is likely to be found in an increasing number of schools. It supports 16 colour text and graphics, a three voice music synthesizer, user defined character fonts, joysticks, paddles, and a light pen. A cartridge containing CBM COMAL-80 Revision 2.0 will soon be available for this microcomputer. This version of COMAL will include commands that emulate LOGO graphics. Packages, which allow for the grouping of procedures, functions, and data structures into special disk files which can be dynamically

invoked from running programs or other packages, are also supported. This would be an ideal environment for the future development of the CASTLE System.

References

Apple Computer Inc. *Apple PILOT: Language reference manual.* Cupertino, California: Apple Computer Inc., 1980.

Burns, P.K. & Bozeman, W.C. Computer-assisted instruction and mathematics achievement: Is there a relationship? *Educational Technology*, 1981, *21*(10), 32-39.

Chambers, J.A. & Bork, A. *Computer assisted learning in U.S. secondary/elementary schools* (Research Report No. 80-03). Fresno: California State University, Centre for Information Processing, 1980.

Chambers, J.A. & Sprecher, J.M. Computer assisted instruction: Current trends and critical issues. *Communications of the ACM*, 1980, *23*(6), 243-332.

Clement, F.J. Effective considerations in computer-based education. *Educational Technology*, 1981, *21*(4), 28-32.

Control Data. *PLATO author language reference manual.* St. Paul: Control Data Corporation, 1978.

Forman, D. *Instructional use of microcomputers: A report on B.C.'s pilot project* (Discussion Paper 03/81). Ministry of Education, Province of British Columbia, 1981.

Freiberger, P. The computer instructor. *InfoWorld*, November 23, 1981, pp. 11-13.

Gleason, G.T. Microcomputers in education: The state of the art. *Educational Technology*, 1981, *21*(3), 7-18.

Godfrey, D. & Sterling, S. *The Elements of CAL.* Victoria: Press Porcepic Ltd., 1982.

Hallworth, H.J. & Brebner, A. *Computer assisted instruction in schools: Achievements, present developments and projections for the future*. Edmonton: Alberta Education, Planning and Research, 1980.

Honeywell Information Systems. *NATAL-II beginner's guide.* Toronto Software Development Centre, 1981. (a)

Honeywell Information Systems. *NATAL-II language specification manual*. Toronto Software Development Centre, 1981. (b)

Honeywell Information Systems. *NATAL-II utilities manual*. Toronto Software Development Centre, 1981. (c)

Hunka, S. Microcomputers in the classroom. *Alberta Printout*, 1981, *2*(3), 8-10.

Hunka, S. & Romaniuk, E.W. *A research and development proposal for the establishment of a computer-assisted instruction facility*. University of Alberta, Division of Educational Reseach Services, 1974.

Hunka, S. et al. *Preliminary specifications of an instructional support system for NATAL-74*. Ottawa: National Research Council of Canada, 1978.

Johnson, J.W. Education and the new technology: A force of history. *Educational Technology*, 1981, *21*(10), 15-23.

Kearsley, G.P. Some 'facts' about cai: Trends 1970-1976. *Journal of Educational Data Processing*, 1976, *13*(3), 1-11.

King, D. Keynote address - Second annual conference of the Alberta Society for Computers in Education. *Alberta Printout*, 1981, *2*(4), 11-14.

Klassen, D. & Solid, M. Toward an appropriate technology for education. *Educational Technology*, 1981, *21*(10), 28-31.

Lindsay, L. *COMAL handbook*. Reston, VA: Reston Publishing Company, Inc., 1983.

Lindsay, P., Marini, A., & Lancaster, M. Survey outlines microcomputer use. *Educational Computing Organization of Ontario Newsletter*, 1980, *1*(4), 27-31.

Luehrmann, A. Computer literacy - what should it be? (Publication unknown - cited by Forman, 1981), August, 1980.

Menashian, L.S. Continuing education resources for electronics-based, high-technology R & D professionals: Part one: Overview. *Educational Technology*, 1981, *21*(11), 11-20.

Moursund, D. Microcomputers will not solve the computers-in-education problem. *Association for Educational Data Systems Journal*, 1979, *13*(1), 31-39.

Petruk, M. *Microcomputers in Alberta schools*. Edmonton: Alberta Education, Planning and Research, 1981.

Rice, J. *My friend - the computer* Teaching guide and activity book. Minneapolis: T.S. Denison & Company, Inc., 1976.

Roblyer, M.D. When is it 'good courseware'? Problems in developing standards for microcomputer courseware. *Educational Technology*, 1981, *21*(10), 47-54.

Romaniuk, E.W. *A versatile authoring language for teachers.* Unpublished doctoral dissertation, University of Alberta, 1970.

Splittgerber, F.L. Computer-based instruction: A revolution in the making. *Educational Technology*, 1979, *19*(1), 20-26.

Travers, J.G. *Development of a microcomputer implementation model: An in-situ, adaptive research paradigm.* Unpublished master's thesis, University of Alberta, 1981.

Voyce, S. *A multilingual-interpreter system for languages used in computer assisted instruction.* Toronto: The Ontario Institute for Studies in Education, 1979.

Westrom, M. *Summary and current status of NATAL-74.* University of British Columbia, Faculty of Education, 1976.

Wise, D. How manufacturers are selling micros to schools. *InfoWorld*, November 23, 1981, pp. 18-19.

Appendix - Glossary

The following items were selected from the glossary in Godfrey and Sterling (1982, pp. 271-279).

access: The ability to obtain information from, or place information into storage.

algorithm: An orderly procedure (akin to a recipe) for obtaining a particular result or solving a problem. Algorithms are often expressed in mathematical terms.

alphanumeric: Alphabetic and numeric characters.

BASIC: (Beginner's All-purpose Symbolic Instruction Code): A compiler or interpreter language that is easy to learn. Used with most time-sharing and minicomputer systems. Oriented toward beginners rather than experienced programmers. Numerous incompatible versions exist, often called dialects: CBASIC,MBASIC,XYBASIC.

bootstrap: A short loader program that loads a more sophisticated loader into memory. That loader, in turn, loads the desired program. The term bootstrap arises from the idea that the computer is picking itself up by its bootstraps. In other words, it progresses from the bootstrap to the loader to the main program itself.

buffer: Memory area in a computer or peripheral used for temporary storage of information that has just been received. The information is held in the buffer until the computer or device is ready to process it. Hence, a computer or device with memory designated as a buffer area can process one set of data while more sets are arriving.

bug: A programming error. Also refers to the cause of any hardware or software malfunction.

byte: In data processing, a sequence of adjacent binary digits (usually eight) operated on as a word, but usually shorter than a word. The value of the bits can be varied to form as many as 2(8) or 256 permutations. So, one byte of memory can represent an integer from 0 to 255 or from -127 to plus 128.

character set: The repertoire of characters that an output device can display or print....

command: A request to the computer that is executed as soon as it has been received. Sometimes this word is used

inter-changeably with the terms "instruction" and "statement". Those terms properly refer to portions of programs and not to commands, which are carried out immediately.

CPU (Central Processing Unit): The primary component of all computer systems. It is responsible for controlling system operation as directed by the program it is executing.

CRT (Cathode Ray Tube) terminal: A type of communications terminal that displays its output on a television-like screen. Synonym of a video terminal.

cursor: A symbol on the display of a video terminal that indicates where the next character is to be located.

disc: A circular piece of material which has a magnetic coating similar to that found on ordinary recording tape. Digital information can be stored magnetically on a disc much as musical information is stored on a magnetic tape. This term is often (and confusingly) used also to refer to a disc drive.

disc drive: A peripheral which can store information on and retrieve information from a disc. A floppy disc drive can store information from a floppy disc and can retrieve that information.

diskette: A small floppy disc in a square plastic envelope commonly either about 13 or 20 cm on a side. See Floppy disc.

disc storage: A type of mass memory in which information is stored on a magnetically sensitive rotating disc. Disc drives are generally both faster and more expensive than paper tape or magnetic tape devices.

editor: A program that facilitates the editing of textual material or computer software.

execute: To perform a computer instruction or run a program.

file: A group of related information records that are treated as a unit. The records may consist of data or program instructions.

firmware: Software stored in read-only memory. Also a synonym for microcode.

flag: A bit whose state signifies whether a certain condition has occurred.

Floppy disc: A slow-speed inexpensive type of memory storage

that uses flexible, or "floppy," discs (or diskettes), made of a material similar to magnetic tape, as opposed to "hard" discs made from rigid materials. It is a convenient method for the "bulk storage" of data, but slower than main computer memory (by 10,000 times) since data is stored in serial form.

hard copy: Information printed on paper or other durable surface. This term is used to distinguish printed information from the temporary image presented on the computer's CRT screen.

hard disc: Disc storage that uses rigid discs rather than flexible discs as the storage medium. Hard-disc devices can generally store more information and access it faster. Cost considerations, however, currently restrict their usage to medium and large-scale applications. Smaller, cheaper units are now coming to market.

hardware: A popular expression used to distinguish the physical parts making up any electronic equipment from the software.

high-level language: Computer language that allows the programmer to write programs using verbs, symbols and commands rather than machine code. Some common high-level languages are: ALGOL, APL, BASIC, COBOL, FORTRAN, NATAL, PL/1, PL/M and SNOBOL.

initialize: To set up the starting conditions necessary for the execution of the remainder of a program. For example, in a program that draws a circle, the initialization might include specifying the radius of the circle. To prepare a diskette so that the computer can later store data on it.

interactive: Said of a computer system which responds to the user quickly - usually less than a second for a typical action. All personal computer systems are interactive.

I/O (input and/or output): A keyboard, a floppy disc and a printer are all I/O devices.

keyword or key word: A word that has meaning in a computer language. See Reserved word.

label: A name comprised of letters, numbers or symbols used to identify a statement or instruction or segment in a program.

language: A set of conventions specifying how to tell a computer what to do.

loop: A program segment that is executed several times in a

row.

menu: A list of options from which to choose.

microcomputer: A computer based on a microprocessor.

microprocessor: A one-chip Central Processing Unit developed in 1971. An integrated circuit that performs the task of executing instructions.

OS (Operating System): A collection of programs to aid a person in controlling a computer. This term is usually used in reference to large computers. A small computer operating system is often called a monitor.

program: A sequence of instructions that permit a computer to perform a task. A program must be in a language that the computer can understand.

programmable memory: Content changeable memory, as opposed to read-only memory (the contents of which are fixed during manufacture). Programmable memory can be both read from and written into by the processor, and is where most program and data are stored. Sometimes called RAM, but this is a slight misnomer.

prompt: A symbol that appears on your computer's display to let you know that it is ready to pay attention to your commands.

RAM (Random-Access Memory): The main memory of any computer. Information and programs are stored in RAM, and they may be retrieved or changed by a program. For some computers, the information in RAM is lost whenever the power is turned off.

reserved word: A word that you cannot use as a variable name, since it has been pre-empted for use in the computer's language. You also may be restricted from using reserved words in other ways as well. Key words are often reserved words. See key word.

response time: The amount of time required for a computer to respond to an input from one of its terminals.

ROM (Read-Only Memory): Memory in which the information is stored once, usually by the manufacturer, and cannot be changed. Programs such as BASIC interpreter (used by most owners of personal computers) are often stored in ROM.

run time: The time at which the program is executed. Also, the amount of time required to execute the program.

save: To store a program anywhere other than in the computer's memory, for example on a diskette or cassette tape.

software: A general term for all programs and routines used to implement and extend the capabilities of the computer: e.g. assemblers, compilers and subroutines. "Software" sometimes means data as well as programs.

structured programming: An attempt has been made to formalize the elements of good programming. These practices have influenced the development of structured languages like Pascal which stress modularity, clear pathways and simplicity.

system program: A program that does not perform actual problem solving but rather is used to control system operations or act as a programming aid.

terminal: A device for communication with a computer. A typical terminal consists of a key-board and a printer or video display.

utility: A frequently used program or subroutine. Utility routines are most often associated with systems programs rather than applications programs.

window: A portion of the computer's display that is dedicated to some special purpose.